# TEMPORAL STREAMS
# PROGRAMMING ABSTRACTIONS FOR DISTRIBUTED LIVE STREAM ANALYSIS APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

by

David B. Hilley

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology
December 2009

Copyright © 2009 by David B. Hilley

**TEMPORAL STREAMS**
**PROGRAMMING ABSTRACTIONS FOR DISTRIBUTED LIVE STREAM ANALYSIS APPLICATIONS**

Approved by:

Professor Umakishore Ramachandran,
Advisor
School of Computer Science
College of Computing
*Georgia Institute of Technology*

Assistant Professor Nathan Clark
School of Computer Science
College of Computing
*Georgia Institute of Technology*

Professor Calton Pu
School of Computer Science
College of Computing
*Georgia Institute of Technology*

Dr. Roger Haskin
Storage Systems Group
*IBM Research*

Associate Professor James M. Rehg
School of Interactive Computing
College of Computing
*Georgia Institute of Technology*

Date Approved: 11 October 2009

*To all my loving family,*

# ACKNOWLEDGEMENTS

"Oh, yes, the *acknowledgements*. I think not. I did it. I did it all, by myself."

– Olin Shivers, `scsh` reference manual

My time at Georgia Tech has been long and challenging, but ultimately rewarding. I can credit my initial graduate school trajectory to several key experiences. I'd like to thank former College of Computing Professor and Associate Dean Kurt Eiselt for introducing me to functional programming in the form of Scheme; he also offered me and some of my friends an enthralling glimpse into the sheer depth of computing as a field in his introductory CS1311X programming class. Early in my undergraduate experience, Jim Greenlee and Bill Leahy challenged me in various classes, and their standards encouraged me to excel. I'd also like to thank Professor Olin Shivers for both his shrewd dissertation advice and for his eye-opening compilers class; he uses his classes as a pulpit to encourage students to pursue advanced research in computer science.

In my second year at Georgia Tech, I started doing undergraduate research with Professor Kishore Ramachandran. Little did I know at the time that I would be a member of his research group for seven more years spanning the rest of my undergraduate and graduate career! I cannot overstate the importance of his support and continued investment in me; his guidance has helped me to mature as a researcher and for that I am truly indebted.

Many members of Kishore's research group, the Embedded/Pervasive Lab (EPL), have also helped me along the way. In particular, I'd like to acknowledge my friends and frequent

collaborators Dr. Hasnain Mandviwala, Dave Lillethun, Nova Ahmed, Junsuk Shin, Dr. Rajnish Kumar and Namgeun Jeong. I'd also like to thank my early CERCS collaborators and mentors, Dr. Matt Wolenetz and Bikash Agarwalla, and all my other lab-mates and friends in the EPL for their encouragement and friendship – Dr. Xiang Song, Dushmanta Mohaptra, Lateef Yusuf, Mungyung Ryu, Hyojun Kim, Kirak Hong and Amit Warke.

In addition to my advisor's research group, I would also like to thank my external collaborators Dr. Kathleen Knobe and Dr. Kenneth Mackenzie, as well as fellow students Matt Flagg and Dongshin Kim for their advice and insight while working together on a common project umbrella. I'd also like to thank my friends and colleagues at GTISC – David Dagon, Robert Edmonds and Paul Royal.

The members of my thesis committee – Dr. Roger Haskin and Professors Nate Clark, Jim Rehg and Calton Pu – have provided invaluable feedback to help focus and strengthen my work. I greatly appreciate their time investment and it is an honor (and rare opportunity) to have the ability to discuss my work with so many experts and deep minds in computer science. Their continued input and guidance has improved the quality of my research.

Additionally, my continuing relationship with IBM has helped in many ways. IBM was gracious enough to award me a graduate fellowship for two years, and my summer internships at IBM in Poughkeepsie and IBM Research Almaden were greatly enjoyable and expanded my outlook. Working on a cutting-edge commercial product and systems research artifact (GPFS) is a rare opportunity for a graduate student. I'd like to thank all of my great IBM co-workers, especially my managers Lyle Gayne and Dr. Roger Haskin, as well as my mentors Dr. Gautam Shah and Dr. Renu Tewari.

Personally, I'd like to thank my parents and sister for their support and love during my long academic journey, as I continually eschewed the real world by earning degrees.

From an early age, my grandparents and parents indulged and encouraged my intellectual curiosities with books, chemistry sets, gyroscopes, prisms and many other educational toys. During my time in higher education, I have had the luxury of living close to my immediate and extended family, and I thank them for their support of my academic endeavors. I'd also like to thank Brian Rutland for his friendship, support and patience during my academic career; I especially appreciate his sympathetic ear, as no one else would have put up with so much complaining. Without his encouragement, I would have long since cracked up.

Finally, this work would not have been possible without funds provided by the NSF, IBM, Intel and Georgia Tech. Over the years, my work has been funded in part by an NSF ITR grant CCR-01-21638, NSF NMI grant CCR-03-30639, NSF CPA grant CCR-05-41079, NSF CSR grant CCR-08-34545 and the Georgia Tech Broadband Institute. The equipment used in the experimental studies is funded in part by an NSF Research Infrastructure award EIA-99-72872 and Intel. My graduate work was also funded in part by a Georgia Tech President's Fellowship and an IBM Ph.D. Fellowship.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF SYMBOLS OR ABBREVIATIONS

| | |
|---|---|
| **CCTV** | Closed-Circuit Television. |
| **CEP** | Complex Event Processing. |
| **DSPS** | Data Stream Processing System. |
| **EBS** | Amazon Elastic Block Store. |
| **EC2** | Amazon Elastic Compute Cloud. |
| **ESP** | Event Stream Processing. |
| **ILM** | Information Lifecycle Management. |
| **IPTV** | Internet Protocol Television. |
| **ITS** | Intelligent Transportation System. |
| **PTZ** | Pan, Tilt, Zoom. |
| **RFID** | Radio-Frequency Identification. |
| **S3** | Amazon Simple Storage Service. |
| **SDMS** | Stream Data Management System. |
| **SLOC** | Source Lines Of Code. |
| **STL** | C++ Standard Template Library. |
| **TCC** | Transportation Control Center. |
| **TLS** | Transport Layer Security. |
| **TMC** | Transportation Management Center. |
| **VaaS** | Video as a Service. |
| **VDS** | Video Detection System. |

# SUMMARY

Continuous live stream analysis applications are increasingly common. Video-based surveillance, emergency response, disaster recovery, and critical infrastructure monitoring are all examples of such applications. These applications are distributed and typically require significant computing resources (like a cluster of workstations) for analysis. In addition to live data, many such applications also require access to historical data that was streamed in the past and is now archived. While distributed programming support for traditional high-performance computing applications is fairly mature, existing solutions for live stream analysis applications are still in their early stages and, in our view, inadequate.

In this dissertation, we present *temporal streams*, a programming model supporting a higher-level, domain-targeted programming abstraction for such applications. It provides a simple but expressive stream abstraction encompassing transport, manipulation and storage of streaming data. The semantics of the programming model are tailored to the application domain by explicitly recognizing the temporal aspects of continuous streams, providing a common interface for both time-based retrieval of current streaming data and data persistence. The unifying trait of time enables access to both current streaming data and archived historical data using the same interface; the communication and storage abstraction are the same – a unified stream *data* abstraction, uniformly modeling stream data interactions.

*Temporal streams* defines how distributed threads of computation interact implicitly via streams, but does not impose a particular model of computation constraining the interactions between distributed actors, targeting loosely coupled distributed systems with

no centralized control. In particular, it targets stream analysis scenarios requiring significant signal processing on heavyweight streams such as audio and video. These unstructured streams are data rich but are not directly interpretable until meaningful features are extracted; consequently, feature detection and subsequent analysis are the major computational requirements.

We also use the programming model as a vehicle for exploring systems software design issues, realizing *temporal streams* as a distributed runtime in the tradition of loosely coupled distributed systems with strong communication boundaries. We thoroughly detail the concrete software architecture and elements of implementation. We describe two generations of system implementations, including the broad development philosophy, specific design principles and salient low-level details. The runtime is designed to be relatively lightweight and suitable as a substrate for higher-level, more domain-specific middleware or application functionality. Even with a relatively simple programming model, a carefully designed system architecture can provide a surprisingly rich and flexibly substrate for upper software layers.

We also evaluate our system implementation in two ways; first, we present a series of quantitative experimental results designed to assess the performance of key primitives in our architecture in isolation. We also use motivating applications to evaluate *temporal streams* in the context of realistic application scenarios. We develop three motivating applications and provide quantitative and qualitative analyses of these applications in the context of *temporal streams*. We show that, although it provides needed higher-level functionality to enable live stream analysis applications, our runtime does not add significant overhead to the stream computation at the core of each application.

Finally, we also review the relationship of *temporal streams* (both the programming

model and architecture) to other approaches, including database-oriented Stream Data Management Systems (SDMS), various stream processing engines, stream programming languages and traditional distributed programming systems and communication frameworks.

# CHAPTER I

# INTRODUCTION

Continuous stream analysis applications are both useful and ubiquitous. Analysis of continuous streaming data is a central component of many programs, both currently deployed systems and conceptualized futuristic ones. Network monitoring, surveillance, robotics, inventory tracking, traffic analysis, weather forecasting, disaster response, stock trading and many other application domains fall under this umbrella. All of these applications have a similar pattern in common: live streaming data is analyzed continuously, and the results of the analysis are used in some sort of feedback loop to direct further analysis and perform external side-effects such as triggering alerts, producing continuous data summaries for human consumption/monitoring or manipulating the environment. We call this class of applications *live stream analysis applications*, because streams are analyzed and consumed live, as the data is produced; since the streams are unbounded and new data is produced perpetually, online ("live") analysis is required. In addition to live data, many such applications also require access to historical data – data that was streamed in the past and is now archived. Non-trivial instances of these applications are distributed, typically requiring significant computing resources (like a cluster of workstations) for analysis.

These applications are challenging to build for many reasons, but ultimately domain- and application-specific analyses are the "heart" of each system. Architectural and structural challenges are largely incidental and could be heavily mitigated by better systems-level support tailored for such applications. Ideally, developers would concentrate on

application-specific analysis code, while other concerns such as stream delivery, resource management and data storage are handled by the supporting infrastructure. A complete solution would consider both lower-level issues (such as storage, transport, resource management, etc.), as well as the higher-level programmer-visible interface to the system – in particular, the abstractions that the analysis code uses to interact with the supporting software. Naturally, higher-level abstractions will lead to simpler, more robust analysis code.

Here we present *temporal streams* – a programming model with semantics tailored to continuous live stream analysis applications. The broad goal of *temporal streams* is providing higher-level distributed programming abstractions for these applications, targeting scenarios requiring significant signal processing on heavyweight streams such as audio and video. This subset of continuous stream analysis applications involves heavyweight input streams of relatively unstructured data (such as video and audio) where feature detection and analysis are the major computational requirements; this is in contrast to systems with a large number of relatively lightweight streams of structured data,[1] exemplified in Linear Road [39] and financial analysis scenarios. Unlike full-stack solutions typified by streaming database work, *temporal streams* is designed to be a lightweight lower-level substrate capable of supporting a variety of higher-level domain-specific functionality – i.e., a slim but domain-targeted building-block, not a top-to-bottom, one-size-fits-all solution.

**Problem Statement:** Current solutions for constructing live stream analysis applications as loosely coupled systems of independent communicating components do not address domain-specific issues of live stream analysis, namely time and historical stream data. This leaves a large gap in the solution space – streaming database systems and compiler-oriented

[1] where the individual data items are directly interpretable, like stock prices or position data

stream solutions all model both stream data interactions and computation and provide full-stack solutions for tightly coupled systems. Current general-purpose distributed programming solutions support loosely coupled, decentralized distributed applications but do not address key issues specific to live stream analysis. In particular, such systems focus on the transport aspects of live stream data, but leave the problem of stream persistence and historical data to separate components. This introduces an artificial distinction between accessing current streaming data and data that was streamed in the past. In addition, the concept of time is fundamental to continuous live streaming data, but such systems leave higher levels of the application stack to deal with the issue of time, treating continuous live streams as an ephemeral flow of ordered bits or data items.

**Thesis:** A real time based representation of continuous streams can enable the straightforward and efficiently realizable unification of both live and archived stream data, easing complexities inherent in distributed live stream analysis applications. Put more succinctly, a real time based representation of continuous streams is 1) useful – easing complexities in distributed live stream analysis and 2) feasible – efficiently realizable. In this dissertation, we describe a programming model called *temporal streams* which provides a simple but expressive time-oriented stream abstraction encompassing transport, manipulation and storage of streaming data. *Temporal streams* defines how distributed threads of computation interact implicitly via streams, but does not impose a particular model of computation constraining the interactions between distributed actors, targeting loosely coupled distributed systems with no centralized control. The unifying trait of time enables access to both current streaming data and archived historical data using the same interface; the communication and storage abstraction are the same – a unified stream *data* abstraction,

uniformly modeling stream data interactions. The recognition of time also eases synchronization.

**Contributions:** The primary contributions of this dissertation are the following:

1. A time-based programming model for modeling data interactions in live stream analysis applications.

   Chapter 3 presents the semantics of the *temporal streams* abstract programming model. It provides a simple but expressive stream abstraction encompassing transport, manipulation and storage of streaming data. The semantics of the programming model are tailored to the application domain by explicitly recognizing the temporal aspects of continuous streams, providing a uniform interface for both time-based retrieval of current streaming data and data persistence. The model defines a variety of expressive *time variables* and a stream synchronization mechanism in addition to data persistence facilities. Chapter 4 presents a set of programming examples to illustrate basic use of the programming model.

2. A runtime architecture realizing the programming model.

   Chapter 5 briefly summarizes a high-level software architecture realizing *temporal streams* as a distributed runtime. Chapter 6 thoroughly details the concrete software architecture and elements of implementation. We describe two generations of runtime prototypes, including specific design principles, selected low-level details and an analysis of potential implementation and architectural enhancements.

3. A targeted system-level quantitative evaluation of the runtime performance.

   Chapter 7 presents a set of targeted quantitative evaluations measuring lower-level system performance in detail. These experiments are targeted benchmarks measuring

specific system components in isolation, designed to assess the performance of key primitives in our architecture.

4. An application-based qualitative and quantitative evaluation exploring implementation properties and runtime performance.

   Chapter 8 evaluates the system in the context of specific application case studies, showing how the system features support different stream analysis scenarios and its achievable performance. We present three realistic application scenarios and describe their context, implementation and evaluation. Each example highlights different properties of the programming model and system. We close with a high-level discussion of a fourth application scenario.

# CHAPTER II

# CONTEXT

> "As such the vision we have for our system is very much in the tradition of systems design. We are focusing on providing a basic set of primitives which are both sufficiently primitive to encourage building arbitrary solutions while being high-level enough to help constrain the complexity and burden of implementation."
>
> – Jonathan Appavoo, Volkmar Uhlig and Amos Waterland [36]

This chapter provides an extended exposition of application context (Section 2.1) and the rationale behind – and need for – *temporal streams* (Section 2.2). The application context is designed to frame the problem space: features of the programming model are driven by particular needs of the provided application scenarios. We follow with a description of the particular personal experiences prompting the development of *temporal streams* (Section 2.3); our own experiences and frustrations constructing live stream analysis applications with existing solutions have influenced the system and guided the design philosophy. In Section 2.3, we also frame the intended scope of our effort. This chapter concludes with an elaboration of the design philosophy that shapes *temporal streams* (Section 2.4) and a limited outline of the remainder of this document (Section 2.5).

## 2.1 Application Context

A myriad of application scenarios involve *live stream analysis* – network monitoring, surveillance, robotics, inventory tracking, traffic analysis, weather forecasting, disaster response and stock trading are just a few common examples. Surveillance is the most broadly accessible example, though admittedly controversial in some contexts. Consider building surveillance as a canonical scenario: an office building may be equipped with hundreds of video cameras located in and around the site, as well as motion, audio and temperature sensors in key areas. Some form of automated baseline analysis runs on all of the available streaming data sources to isolate streams of interest, such as video cameras currently sensing motion. The smaller set of "interesting" streams are then targeted for more computationally intensive analysis like object tracking, face detection/recognition, or even the baseline analyses at higher levels of detail. Currently, a realistic system would probably direct the attention of security personnel, presenting them with a small subset of *streams of interest* picked automatically. In a more ambitious and futuristic scenario, computationally intensive and targeted analysis might replace most manual human tasks. Traffic monitoring by the Department of Transportation (DOT) in various US states, citywide CCTV security cameras in London and Chicago (IBM's Smart Surveillance System – S3 [182]), and dock-/port/airport/utility security systems (with components provided by companies like Vistascape, ObjectVideo, IntelliVision, Siemens, Sarnoff Corporation, Verint and Cernium) are all examples of real, currently deployed systems with high similarity to the office building security scenario presented above. These applications are often grouped under the umbrella of *situational awareness* systems.

Consider a team of autonomous robots as another application scenario: each robot is equipped with many sensors and actuators and will continuously monitor streamed sensor

input, perform analysis on the data, and potentially react to the results of analysis including communicating and coordinating with its peers or manipulating the environment. Input streams are transformed into higher-level feature information by continuous analysis. These higher-level feature streams are then used as inputs to online mapping and planning algorithms. A slight variation of this scenario involves a team of robots with some additional external command and control centers. The command and control centers can host more extensive computational resources free of the robots' power and mobility constraints. For robustness, however, the robots cannot simply be slaves to an application running remotely at the command center – some control and analysis must run locally on the robots in case the command center becomes unreachable or if the results of analysis are highly latency sensitive. In this manner, the application must span a more dynamic environment than traditional HPC approaches assume.

A third application scenario is a TV content *recommender system* [170]. An example application could make recommendations about current programming of interest to a specific user by performing real-time media content analysis. One compelling use of live stream analysis in this context is the generation of dynamic content metadata/content profiles for live programs. "TV guide" systems use static metadata about the topics, guests and themes of pre-recorded programs. Live programming, such as news/talk shows, sports events and other up-to-the minute broadcasts have very limited pre-generated static metadata since the content is not known a priori. Live analysis provides the potential for more descriptive dynamic metadata generation, as well as adding a temporal dimension to metadata relevance.[1] TV Watcher [103] is an example application using this strategy: live

[1]This is certainly possible with pre-/human-generated metadata as well – for instance by tagging with time ranges – but such fine-grained detail is not commonly available.

stream analysis on television broadcasts' closed captioning streams generates temporally relevant recommendations based on dynamic program content. More ambitious extensions could utilize video or audio analysis to extract higher-level feature information.

These scenarios, while idealized, highlight patterns of communication and computation common in live stream analysis applications.

## *2.2 Rationale*

Instances of *live stream analysis* applications are increasingly prevalent – the number of continuous data feeds is growing at an impressive rate. While many critical applications involve continuous and computationally intensive analysis on live streaming data, many such applications also require some access to historical data (either data that was streamed in the past or some summarized form of prior data). While distributed programming support for traditional high-performance computing applications is fairly mature, existing solutions for live stream analysis applications are still in their early stages and, in our view, inadequate.

Existing solutions for constructing such applications tend to fit into two broad categories: 1) "stream database" or "stream processing engine"-style systems or 2) general-purpose distributed programming systems. The former category has centrally managed and controlled execution, while the latter does not impose a particular computational model on applications, only modeling data interactions. The latter support loosely coupled systems of independent communicating components with no centralized control. To the best of our knowledge, no prior system has provided a unified abstraction for both transport and storage of live streams as a distributed programming primitive. Our approach provides simple and efficient programming idioms for dealing with distributed stream data, explicitly recognizing the semantic importance of time in live streams. See Chapter 10 for a thorough

discussion of the relation of our system to prior work.

Our approach is in providing a simple but powerful programmatic interface to continuous live data streams as distributed data structures called *temporal streams*. This abstraction provides a uniform interface for both time-based retrieval of current streaming data and data persistence. It fits between very high-level and heavy-weight solutions like full databases with query languages and lower-level non-stream oriented distributed communications facilities typically used for distributed applications (MPI, RMI, etc.) plus separate storage facilities. This middle ground in the design space for continuous streaming data is roughly analogous to solutions such as Distributed Data Structures [95], BerkeleyDB [155] and Boxwood [132] for non-streaming data. The *temporal stream* provides first-class recognition of time, which is a critical distinguishing aspect of continuous live streams over other streaming data; the tailoring of the abstraction to the problem domain makes live stream analysis applications more straightforward to build. Since analysis code must interact with the stream transport layer to retrieve and produce data, a persistent storage mechanism should interface transparently with the stream transport abstraction; this eliminates a programmer-visible artificial distinction between "live" data that is currently being streamed and stored data that was streamed in the past.

## *2.3 Impetus and Focus*

The ideas behind *temporal streams* evolved both out of my own personal experiences and the experiences of various members of the Embedded/Pervasive Lab in attempting to implement live stream analysis applications on a variety of substrates, including Stampede [167], D-Stampede [31], MPI [87], Java Message Service [99], Sun RPC [186], RMI [11], tuple spaces and various custom messaging layers. We encountered certain development "pain

points" which *temporal streams* is designed to address; in particular, the challenges of dealing with the notion of time across the system as well as storage and access to historical stream data were near universal. Management of computation was often an issue, but we found that the computational requirements varied significantly between application domains. Therefore *temporal streams* does not impose a specific computational model; it only models stream *data* interactions, allowing applications to build domain specific computation management with it and integrate *temporal streams* into systems with many different requirements.

My experience implementing TV Watcher [103], Media Broker [144]/MB++ [127] and various application kernels on Streaming Grid [32], as well as maintaining D-Stampede [31] provided insights about the nature of such applications and the effect of the higher-level requirements on the system-level support infrastructure. As my colleagues have implemented RF$^2$ID [33], ASAP [180], and various other live stream analysis applications, we have compared experiences and this feedback has also shaped *temporal streams*.

**Focus:** The development pain points mentioned above are critical issues affecting applications in the live stream analysis domain. *Temporal streams* focuses on addressing these particular problems, which are directly and elegantly handled by a time-based stream representation. Naturally, many other important issues arise in live stream analysis applications, but the scope of this work is limited to a specific subset of issues. Problems such as fault tolerance, management of computation and security are obviously important in any distributed system, but they are considered here specifically where they intersect with our system's focus. For example, our global stream metadata is replicated for fault tolerance, and streams may also be replicated for availability and performance. Also, our system only handles

computation in limited circumstances – namely providing transformative functions for persistent data on secondary storage. Other considerations, like security, are largely discussed in the context of future work. In this manner, our work does not attempt to address all concerns inherent in the target application domain, but instead provides a focused, elegant solution for a set of important core issues.

## *2.4 Design Philosophy*

The quote at the beginning of this chapter articulates the design philosophy that guides *temporal streams* – both the programming model and software architecture. The goal is providing a simple primitive which is high-level enough to capture the essence of a particular problem while still remaining low-level enough to be small, efficient and widely applicable. Small primitives are often quite flexible, despite their simplicity: although the domain-specific, time-based temporal stream abstraction can be viewed as a distributed communication mechanism, it is actually a more general stream *data abstraction*, naturally admitting stream persistence/historical data storage due to its use of time-based indexing. Balancing the cost, generality, power and level of abstraction is a judgement call, but we believe that *temporal streams* occupies a balanced middle ground.

This dichotomy is often seen in the systems community – consider the recent proliferation of non-relational distributed key-value stores (Facebook's Cassandra [117], Google's Bigtable [64], Amazon's Dynamo [72]) in lieu of traditional relational database management systems. A simple but well-designed primitive often provides 90% of the widely useful functionality without sacrificing performance or adding significant complexity. There is no doubt that a full database and query language is an immensely powerful tool, but much of that power is unused in a typical application. In addition, a simple programming model

is easier to learn and understand.

While the *temporal streams* programming model presents a clean and relatively simple abstraction, a carefully designed system architecture realizing that primitive can provide a surprising level of flexibility to its users. This can be bolstered with judicious layering and by providing hooks into key system-level components, enabling powerful extension while staying within the general framework set by the programming model. In addition, there is tremendous system-level value in presenting abstractions high-level enough to capture the essence of a particular problem – the system then has a unique vantage point from which it can observe higher-level application behavior and adapt.

That briefly summarizes the design philosophy behind the programming model and some implications of the interplay between the programming model and the actual system design philosophy. For more on the concrete and lower-level system design philosophy, see Chapters 5 and 6, where these issues are discussed in context of specific software architectural details.

## *2.5 Roadmap*

This rest of this document begins with a description of major facets of the *temporal streams* abstract programming model in Chapter 3. Chapter 4 presents a set of Java and C++ examples using the programming model. Although the code examples make some assumptions about how the abstract programming model is implemented and exposed to the user (i.e., as a library-based API), it is presented directly after the programming model chapter in order to concretize and reinforce the programming model's basic concepts. Chapter 5 briefly summarizes a high-level software architecture realizing *temporal streams* as a distributed runtime. Chapter 6 details the concrete software architecture and implementation

of *temporal streams*. Chapter 7 presents a set of targeted quantitative evaluations measuring lower-level system performance in detail; Chapter 8 evaluates the system in the context of specific application case studies, showing how the system features support different stream analysis scenarios and their achievable performance. Chapter 9 provides a short retrospective discussing the evaluations and overall system properties at a high level. A survey of related work is presented in Chapter 10. Finally, Chapter 11 concludes and presents a few potential avenues of future exploration.

# CHAPTER III

# PROGRAMMING MODEL

Here we present a distributed programming model for live stream analysis applications providing communication and storage abstractions for *temporal streams*. Temporal streams are a key feature of live stream analysis applications – as a concrete example, think of a video camera feed: the stream is unbounded and produced at a finite rate, and the data items (video frames) are temporally ordered. Each frame represents some sampled interval of time based on the frame rate (e.g., a frame may represent 1/30th of a second). Event streams and other aperiodic streams may not have fixed output rates, but trigger based on certain environmental conditions, like a temperature sensor sending an alert when a threshold is reached. In both cases, data items are associated with specific time information. All "live" streams have a natural relationship with time (wall clock time) – time is really the common unifying trait across all live stream analysis applications. Time is the dimension along which all continuously produced streams vary. Broadly, our model of a stream is a time-indexed sequence of discrete data items; each data item has a timestamp and spans a time interval ending with the timestamp of the next item.

The benefit of this model is that it provides a higher-level stream abstraction, which fits at the intersection of an application's manipulation of data and stream transport. By supporting first-class recognition of time, it provides a more natural way to write analysis code that deals with temporal streams – similar to a tuple model in streaming database work, and higher level than general-purpose distributed programming mechanisms appropriate

for high-volume data transfer. Since the fundamental stream abstraction is a time-indexed data structure, it can be used both as a communication mechanism and a data storage interface – a unified stream *data* abstraction, uniformly modeling stream data interactions. Rather than managing and buffering an ephemeral, linear flow of data, the application can access stream data in terms of higher-level time information.

## *3.1 General Paradigm & Overview*

Broadly, an application within the framework of this programming model involves distributed threads of computation communicating implicitly via distributed data structures. In our programming model, a temporal stream is represented by a *channel*, which is a distributed data structure encompassing an interface for both transport and manipulation of streaming data; each channel presents a time-indexed sequence of discrete data items (such as video frames) and analysis code retrieves data items by specifying time intervals of interest. Applications interact with channels by means of *get* and *put* operations. Figure 1 shows a simple example of this paradigm. Two threads place video frames into separate channels used by feature detector threads. The results of the feature detection process are placed into other channels for a higher-level analysis thread to use.

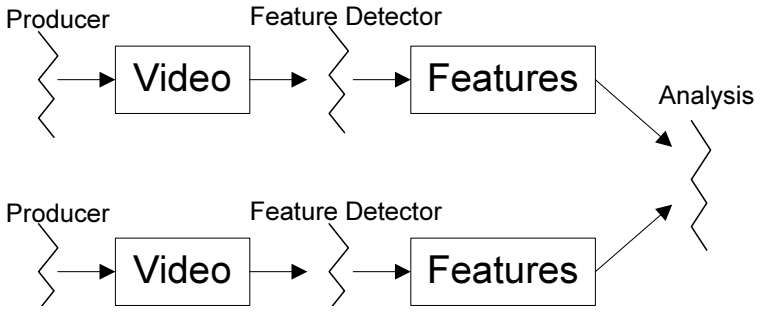



**Figure 1:** Conceptual Application

Channels store discrete items sequenced by *wall clock timestamps*, and they are used to

connect producers and consumers in arbitrary N-to-N configurations. Items of interest are fetched based on the associated timestamp information. Storage handling in channels is automatic and the programmer can choose from several sets of garbage collection semantics (see Section 3.6) on a per-channel basis; programmers can also request stream persistence onto secondary storage (Section 3.7). The programming model also provides facilities for inter-stream synchronization based on temporal information (Section 3.5). Time is the primary attribute used to manipulate and identify items in a channel and is discussed in detail in Section 3.3. Temporal streams also provides simple naming and discovery services for both peer endpoints and channels (Section 3.8).

## *3.2 Streams*

Fundamental aspects of our programming model revolve around the concept of *streams*, which we define as any continuous sequence of temporally ordered data items with no requirements of fixed periodicity. One can further classify streams as *periodic* and *aperiodic*. Periodic streams produce items at an approximately fixed rate, such as a stream of video at 30 frames per second. Aperiodic streams do not have a fixed rate, and typical examples of aperiodic streams are produced in response to events. For instance, a motion sensor may produce alert events every time a certain level of motion is detected. Any aperiodic stream can be turned into a periodic stream by simply sampling at regular intervals and producing "no data"/"no event" items when the aperiodic stream would not have produced anything. Data items in a stream may also have a defined *duration*. For items in a conceptually continuous stream, such as video or audio, the duration would simply be the inverse of the production rate (33.3 milliseconds for video at 30 frames/second), because each item represents a discrete sampling of some continuous data. For other streams, the item duration

may signify the length of time an item is "valid." For the motion sensor example, an item duration may signify a time window during which no new alerts will be signaled. The concept of an item duration may not be intuitively meaningful for some aperiodic event streams.

## *3.3 Real Time*

Items in a channel are indexed by wall clock timestamps, which we also refer to as "real" time (in contrast to virtual time). Although wall clock time is often a perilous issue in distributed systems, it provides the most natural mechanism for programmers dealing with streams and synchronization in live stream analysis applications. The alternative, virtual time [111], is used heavily in discrete event simulation, but can impose awkward constraints on live stream analysis applications. For example, media streams use the concept of real time to synchronize different components of coherent streams (e.g., separated video and audio streams corresponding to the same television program); one cannot derive the frame rate of a video stream from virtual time information alone.

Using real time requires accurate local clocks on all of the participating content producers, which is not an unreasonable burden. NTP [143] is widely deployed and can keep hosts over the Internet synchronized with high precision. NTP also provides facilities for estimating clock skew. The IEEE 1588/Precision Time Protocol (PTP) [24] standard can synchronize LAN-connected systems to microsecond precision in software. For limited devices, more lightweight techniques like Reference Broadcast Synchronization (RBS) [81] or the Flooding Time Synchronization Protocol (FTSP) [135] can be used; alternately, the synchronized clock requirement can be mitigated for limited devices by providing producer proxies on more capable hardware.

## *3.4 Channel Semantics*

Each data item in a channel is associated with a timestamp specified when the item is placed into the channel: we call this timestamp the *production time* of the item because it is typically set to the current wall clock time when the item is placed in the channel by a data producer. Items are ordered in a channel chronologically by production time. In this manner, each item naturally defines a time interval bounded below by its production time and above by the production time of the next item. When an item item is the newest available (i.e., there is no next item), *now* is always the upper bound of the interval. See Appendix A for a quick set-theoretic formulation of channels.

**Timestamped Items:** By default, the timestamp of an item is the current time when it is placed into a channel, but explicit user-provided timestamps are also used in many circumstances; for instance, a computation that transforms a media stream into a feature stream would retain the original timestamps from the media stream so there is a natural mapping between both streams. Fundamentally, this mapping facilitates *synchronization* – establishing temporal correspondences between two or more streams. Since more than one producer may place items into the same channel, multiple items may have identical timestamps, although this is somewhat uncommon on platforms with high granularity timers (timestamps are currently at least microsecond resolution). To accommodate multiple items with the same timestamp, one can alternately view a channel as a sequence of buckets ordered by timestamp, where each bucket contains at least one item.

**Item Retrieval:** Items are retrieved from channels by specifying a time interval containing the items of interest. Since wall clock time is perceived as continuous, programmers will tend to work naturally with time intervals; some intervals are explicitly specified, but

others may be specified using special *time variables*. For example, "5:30pm to 5:35pm" has concrete times as upper and lower bounds, while "the last thirty seconds," is implicitly specified with the current time as an upper bound and "the current time minus thirty seconds" as the lower bound. Another common idiom is retrieving the oldest item newer than a certain bound (typically the last item retrieved).

**Time Variables:** Several special *time variables* are specified for use in the construction of intervals. The special time *now* represents the current time and is always the upper bound of the interval subsumed by the most recently produced item of a given channel. When a newer item is placed into the channel, the time at which it was added becomes the concrete upper bound for the previous item. Other common time variables include *newest* and *oldest*, which are the production times of the most recent and oldest items in a given channel respectively. *newest-after* is a special variable specifying the newest item in a channel only when the newest item is more recent than a given timestamp. Note that this form of predication allows the expression of certain common producer/consumer patterns without requiring a channel to maintain any state on behalf of consumers – for instance, a consumer can ask to wait for the next item it has not yet seen without the channel maintaining any state about which consumers have "seen" various items. All of the necessary state is explicitly pushed into the time variable. Time variables can also be offset by concrete amounts, such as "*newest* minus thirty milliseconds." Table 1 provides a summary of some available time variables.

**Get/Put Semantics:** The programming model does not specify whether *push* or *pull* semantics are used for the actual transfer of remote items, but the conceptual framework

specifies semantics in terms of *get* and *put* operations on items for consumers and producers, respectively. The basic stream operations are 1) put($i, t$) – put data item $i$ on the stream with timestamp $t$ ($t$ defaults to the current time); and 2) get($l, h$) – get items falling within the interval $[l, h)$. In a concrete implementation, there are many different call variations (e.g., to provide get item limits, multiple item puts, blocking/non-blocking semantics, etc.), but all are fundamentally *get* or *put* operations.

When a client performs a *get* operation on an interval, the system will provide all items contained fully or partially within that interval starting with the first full item (inclusive or exclusive bounds are user-specifiable). Figure 2 depicts an example in which a *get* on the specified interval will return the shaded items. Since the runtime has no application-specific knowledge of the data in channels, it cannot subdivide an item, but the application may be able to do so if such an operation is meaningful.[1] Consequently, a *get* operation will provide whole items partially contained within the specified interval, or the user can optionally request only the items fully contained within the interval – this would be useful in the case of very large items that are also indivisible.

**A Note on Time Variables Definitions:** Note that the definition of time variables actually depends on the exact time semantics of channel *get* operations. Previously, we have stated that the *get* operation "will provide all items contained fully or partially within that interval starting with the first full item." With these semantics, the definition of *next* is as simple as incrementing a given timestamp by the smallest discrete timestamp granularity. This works because if item $i$ has a timestamp $t_i$ and we ask for the interval starting with *next*($i$) (i.e., $t_i + \epsilon$), we will not include item $i$ because we start with the first *full* item, which

[1] For example, chunks of *isochronous* sampled data can typically be subdivided in some meaningful way.

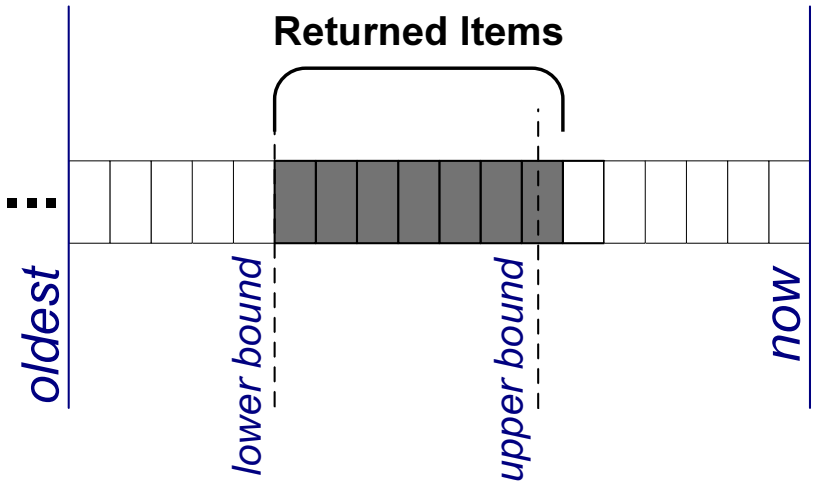



**Figure 2:** Time Interval Example

will be *i*'s successor if one exists.

If the channel *get* semantics did not start with the first full item and instead started with the first item partially intersecting the interval, we would have to modify the definition of *next* to actually find the timestamp of the subsequent item in a given channel. In this case, *next* would require evaluation based on a channel's current contents, rather than being a simple increment.[2] This illustrates that the definitions of variables are dependent on the exact time behavior of channels. From the perspective of a user, however, the underlying definition of the time variables does not matter as long as they behave according to their abstract behavior (i.e., *next* provides access to the next item). We can vary the time behavior of channels and maintain the correct time variable semantics by changing their definitions.

[2]Note that the current semantics would require this kind of content-dependent definition to define a *previous* time variable; the current semantics are a result of the assumption that *next* is commonly used.

**Table 1:** Time Variables

| Variable | Parameter(s) | Definition |
|---|---|---|
| *now* | - | current time |
| *newest* | `ch` | timestamp of newest visible item in channel `ch` |
| *oldest* | `ch` | timestamp of oldest visible item in channel `ch` |
| *newest-after* | `ch`, `ts` | timestamp of newest visible item in channel `ch` if the item's timestamp is after `ts`, $\infty$ otherwise |
| *next* | `ts` | `ts` + $\epsilon$ (the smallest timestamp > `ts`) |

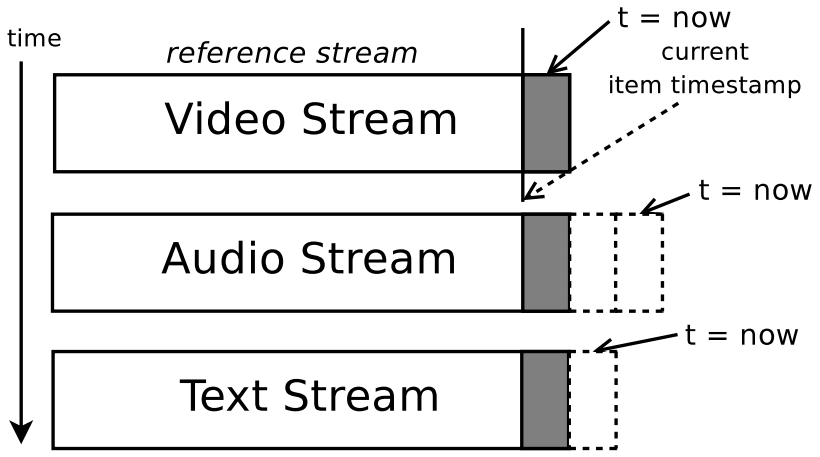
The parameters column shows factors affecting a time variable's definition – for example, *newest* refers to the timestamp of the newest item in a particular channel, so its definition varies depending on which channel we use as a reference point. However, note that this column includes both implicit and explicit parameters – when the programming model is actually used in practice, the channel parameter is implicitly specified by operation in question (e.g., "get *newest* from channel *X*"). On the other hand, timestamp parameters are generally explicit: "get *newest-after* `ts` from channel *Y*." See Chapter 4 for programming examples in context.

## *3.5 Synchronization and Channel Groups*

Synchronization is an important concept for applications dealing with data streams. Frequently, video and audio corresponding to the same logical program stream will be demultiplexed into separate, atomic streams for processing (for example, separate audio and video feature detectors). It is important to provide the programmer with a mechanism to synchronize these streams. The most easily accessible example of stream synchronization is synchronizing audio and video for playback but, more broadly, stream synchronization is about setting up a temporal correspondence between two or more streams – any analysis that uses more than one stream as input will need to synchronize the input streams to get a coherent view of the data. Since the process of stream synchronization is application and content-specific, the programming model does not explicitly synchronize content in any manner that is dependent on the type of data. It instead provides high resolution timing information and support for item-level timed synchronization of multiple channels.

**Reference Streams:** In order to synchronize items across several channels, a client must perform a *get* operation from the same time interval on each channel. As a common example, a programmer may wish to get the item corresponding to *now* (or some other time variable) from several channels; it is not appropriate to simply retrieve the item corresponding to *now* from different channels in sequence because the value of *now* is changing between calls. Instead, the client needs to get an item corresponding to *now* in a single channel and then use the real timestamp value as a reference for *get* operations on the other channels in order to receive a coherent cross-section of items from the same time interval. The process is depicted in Figure 3. The stream that the other streams use as a synchronization reference point is called the *reference stream*. If a single thread is performing all of the gets, the programmer can perform this form of synchronization explicitly by getting an item from the reference stream and using the associated timestamp for subsequent gets; the programming model provides group *get* operations on multiple channels using a reference stream for this purpose. However, when consumers in separate threads need to perform the same operation on a set of channels, extra inter-thread communication would be required for synchronization. In order to alleviate such complications, temporal streams provides system-level support for this manner of synchronization implicitly using an abstraction called a *channel group*.

**Channel Groups:** A channel group is an abstraction which conceptually controls the visibility of items in a set of associated channels without modifying the actual items available. Since the visibility information is associated with the channel group and not its constituent channels, a channel can belong to many groups simultaneously. Channel groups can be created and destroyed dynamically, and they are simply a collection of channels with a designated reference channel. The result of creating a channel group is a new set of channel




The "now" labels depict what *now* would specify if the items were retrieved in sequence from the channels without a grouped operation. The shaded items form a synchronized interval.

**Figure 3:** Reference Stream Pattern

descriptors: each channel descriptor refers to the original input channels, but *get* operations on the new descriptors will be synchronized with the reference stream. Channel group synchronization is performed by controlling both the visibility of data items and the interpretation of time variables to reflect the reference stream's state.

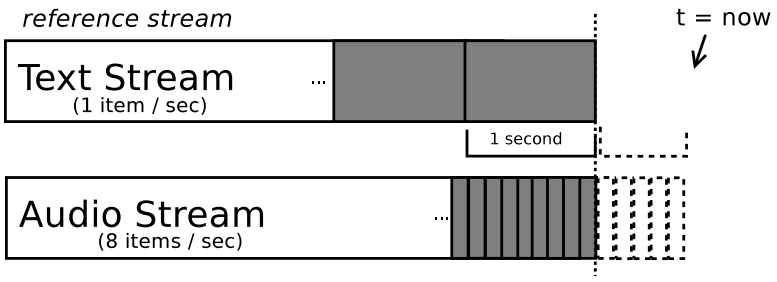
Figure 3 also depicts the conceptual result of creating a channel group on three streams. Items newer than the newest item in the reference stream are unavailable when getting from the channel group descriptors. The hidden items are still in the channel and available through the original, ungrouped channel descriptors. The programmer may also want to dynamically synchronize with the channel that lags the furthest behind at any given moment. In other words, the reference stream for the channel group cannot be set to a specific channel a priori, but instead must be determined by considering the newest item $i$ from each channel and choosing the channel with the oldest $i$. For this purpose, there is a special reference stream identifier *oldest* which refers to the channel with the oldest current item.

**Buffering:** Channels in any given channel group synchronize to the newest item in the reference channel for that group, making the corresponding items in the other channels visible. As mentioned earlier, each item in a channel has a specific timestamp associated with it, and an item spans an interval from its timestamp (a lower bound) to the timestamp of the next item (an upper bound). Since the upper bound for the newest item is not yet known, the lower bound of the newest item $l$ is used for synchronization by default. All items with timestamps $< l$ are visible, so consequently the newest item in the reference stream is hidden. This buffering is necessary since we have no knowledge of the next item's timestamp.

The programmer may additionally provide an implicit estimate of the next item's timestamp in the form of item durations, which provide an upper bound for the newest item. When durations are provided, the system also allows the programmer to choose to use the upper bound for synchronization where appropriate.[3] In conceptually continuous streams, the duration of items is directly derived from the rate of production of items: for example, a stream of video produced at 30 frames per second would assign each item a duration of approximately 33.3 milliseconds. Figure 4 shows an extended example of a channel group reference stream including item durations and hidden items. Text items are produced at the approximate rate of one item per second and have a duration of one second, while audio samples are produced at the rate of eight per second and have a duration of 125ms. If item visibility is synchronized to the production of text items, the shaded items will be available. Several audio items newer than the upper bound of the newest text item have

[3]This is appropriate in situations where one or more producers have some implicit buffering – for example, a video producer whose video frame timestamps represent a time before frames are captured and sent. For there to be any meaningful synchronization, there must be some buffering in the system.

been produced, and they will be made visible as soon as a corresponding text item is produced. Currently, the programming model does not admit temporally overlapping items, so the effective duration of an item is also bounded by the production of the next item (the minimum of the specified duration and the difference in time between items). It is unclear whether a single stream with temporally overlapping items is useful.




"now" represents the current time, and shaded items are "visible" when synchronized using the text channel as a reference stream.

**Figure 4:** Item Duration Example

**Summary:** Channel groups are a mechanism for coarse time-based synchronization of streams. This form of synchronization is not necessarily a direct substitute for application-specific, finer-grained/precise synchronization, but it allows synchronization at item-level granularity. The primary purpose is to keep cooperating live analyses of related streams roughly synchronized. One can also view channel groups as a kind of barrier – instead of waiting for threads/processors to arrive, however, a group of consumer threads are waiting for data to arrive. Waiting for the data to arrive can be analogized to waiting for the group of producer threads generating that data to arrive. In the case where the reference stream is *oldest*, the consumer threads are basically waiting for corresponding data to be produced by all of the producer threads before continuing: this is analogous to a "stream barrier"

operator provided by some declarative stream processing systems [88].

## *3.6 Garbage Collection*

The programming model supports several different garbage handling strategies, and the policy can be set on a per-channel basis (at channel creation time); the philosophy underlying all options leans towards simple solutions not requiring the calculation and constant upkeep of global state.

**Currency/time window-based:** The default and primary option for garbage handling is based on buffering time windows for channels: for instance, the programmer might specify that a time window of thirty seconds is needed for channel $c_1$ and items older than thirty seconds will be discarded automatically. In a system dealing with continuous streams, an important factor is *currency* – a measure of how current data is. The buffering window is an implicit specification of desired currency (a lower bound) and should be a natural way to specify garbage handling for many types of media streams. The bound itself could also be dynamically adjusted or calculated based on the number of consumers (and possibly other factors such as estimated end-to-end latency).

**Other GC models:** In addition to the bounded currency garbage handling model, several other simple strategies can be provided for flexibility. One strategy is simply bounding the total number of live items, where the oldest items are pushed out by newer ones. In some circumstances, applications may wish to constrain channel capacity by limiting the total data size of all items in a channel; in this case, the limit is obeyed by reclaiming the oldest items (however, as with cache-replacement policies, size-limited metrics have to be considered against the possible variance in data size between items). One might also envision uses for various hybrid policies, such as a fixed limit on items plus a lower

currency bound, where a producer would block putting new data into a full channel until the oldest item passes the lower currency bound. Finally, we also provide a simple reference counting model for situations where the number of consumers is known a priori.

## *3.7 Stream Persistence*

Since some live stream analysis applications may also need to store and retrieve historical data for trend analysis, a persistence mechanism that fits within the temporal stream model is a useful feature. The programming model as described so far is useful for a non-trivial subset of applications: individual running analyses may store all data internally in memory, or publish summarized information as just another live stream of data for input to other analysis code. Applications in some domains, however, such as network monitoring, financial analysis or surveillance may require that streams be persisted in some form. For instance, a surveillance system might store historical video streams for some predetermined time period (e.g., two weeks) in a degraded form (lower quality than live streams).

**Benefits:** Integrating the persistence mechanism directly into the programming model makes the model more widely applicable and reduces complexity. Providing a uniform stream abstraction that handles live and stored data can also avoid an artificial distinction between data that is currently available in streams (a window of recent data) and data that was streamed in the past but may now be archived. The ability to talk about historical data also elevates the temporal stream abstraction from a communication abstraction to a general-purpose *data* abstraction, uniformly modeling stream data interactions. While the same abstraction should provide seamless access to both types of data, information about the source of data should still be made available to the programmer as the difference in access time, data quality or data representation can be significant. From a programming

perspective, eliminating unnecessary non-uniformity is often desirable as it can make applications simpler to construct and less brittle in the face of change.

**Flexibility:** In addition to the above concerns, the system should provide for flexible persistence policies in order to support a wide range of live stream analysis applications. "Persistence policies" could include how data items are mapped to a persistent form (e.g., *as is* or compressed/degraded/summarized), how persistent streams are stored (storage backend), when data items are stored (immediately or lazily upon garbage collection), and various information lifecycle management (ILM) issues (redundancy, how free space is reclaimed, hierarchical secondary storage, etc.). Some of these policies are enabled by the storage backend, while others are higher-level concerns.

### 3.7.1 Persistence Interface

Our high-level persistence interface is directly enabled by extending the time-oriented channel interface – a channel can now be marked by an application as persistent (at creation time or later). Persistent channels empower the application programmer in the following ways: 1) items are automatically committed to persistent storage with related time-stamp information, 2) time intervals for retrieval of items may now span both live and persistent items, and 3) the system provides a built-in mechanism to degrade or change the format of channel items as they transition to secondary storage. Retrieval of stored items is also a transparent extension of the existing channel interface: "live" items are normally retrieved from a channel by specifying a time interval of interest. With persistence, if the time interval includes stored items, they are retrieved and returned along with any "live" data items included in the interval. Figure 5 depicts a *get* operation with an interval spanning live and stored data. Other high level interface decisions are described below.

**Get interface:** The application may optionally constrain a retrieval operation to adjust for the difference in latency of access and potential data format differences of stored versus live items. The options are as follows: 1) `ANY` – any items, live or stored; 2) `LIVE` – only live items, 3) `STORED` – only stored items; 4) `ANYSPLIT` – return live items and load stored items from disk in the background, caching them in a temporary in-memory cache for a subsequent *get* operation (allowing live items to be used while stored items are loading). An `ANYSPLIT` *get* operation returns the live items first and a *ticket* which can be used to retrieve the loaded stored items a subsequent call.

**Per-stream data representation:** An application can also control how items are mapped to a persistent form. Some may wish to degrade the quality of items, reduce the number of items or otherwise change their format. An application can provide a *pickling handler*, which is responsible for mapping items to their persistent representation (defaulting to the identity function). For example, a video channel's handler may JPEG compress video frames or reduce the image resolution. In addition to one-to-one item mappings, the pickling handler can take N items and produce a single item to store: for example, a video channel's handler may halve the frame rate by averaging two consecutive frames, or an event channel's handler may transform thirty one-second events into some sort of digest. When N items are mapped to one item, the original timestamp information is retained, so identical *get* requests will operate similarly on live and stored data. That is to say, if two items are mapped to a single stored item, it will span the combined time interval of the original items. As a direct extension of this functionality, an application might provide multiple handlers with varying levels of disk usage versus processing time and the runtime could automatically switch based on system-level cues.

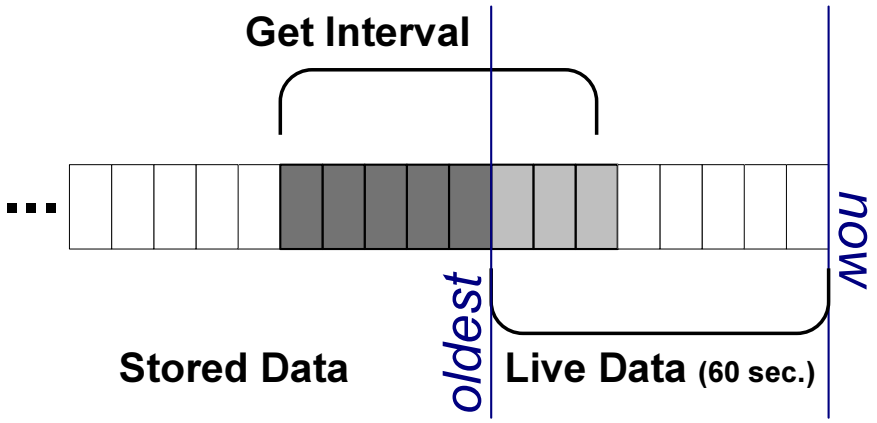



**Figure 5:** Get operation spanning stored and live data

**Per-item persistence control:** In addition to per-stream control via pickling handlers, per-item control is possible: a data producer may mark an item placed into a channel with the `NOPERSIST` flag. This will cause the persistence mechanism to ignore it and the item will disappear for good when it is garbage collected.

**Persistence timing:** An application may also control whether items are persisted immediately or when they are about to be garbage collected from the in-memory "live" stream: *eager persistence* versus *lazy persistence* (the default is the former).

All-in-all, the programmer visible interface to a channel is essentially unchanged – put$(i, t)$ and get$(l, h)$ still operate as before, but the potential span of items available in a channel now includes historical data rather than just a window of current live data. Put takes an optional `NOPERSIST` flag and get takes an optional `ANY`, `LIVE`, `STORED` or `ANYSPLIT` modifier (`ANY` is the default).

## *3.8 Other Features*

The programming model also defines abstractions other than channels for control data or messaging between distributed components. Each client has a mailbox to receive messages from other clients. Simple distributed queues are also provided, and they can use any of the non-time based garbage collection mechanisms (in fact, a mailbox is simply a FIFO queue where each item has a reference count of one). Also provided are naming, registration and discovery services for channels and "participants" of the computation (producers or consumers of data in channels).

# CHAPTER IV

# USING THE PROGRAMMING MODEL

To recap, the programming model presents a temporal stream abstraction called a *channel*, which is a distributed data structure encompassing an interface for stream data interactions; each channel presents a time-indexed sequence of discrete data items (such as video frames) and analysis code retrieves data items by specifying time intervals of interest. Applications interact with channels by means of *get* and *put* operations. A variety of expressive *time variables* are provided to allow *get* operations to specify intervals such as "the most recent 10 seconds" of video data. Multiple streams can be virtually synchronized, and the system also provides for time-windowed garbage collection of items – for example, an application could specify that a particular channel should keep thirty seconds worth of data, and items older than that may be reclaimed. Stream data may also be persisted to stable storage automatically and retrieved transparently with flexible application-controllable persistence policies.

## *4.1 Basic Examples*

In order to concretize our discussion of the programming model, here we present some simple code examples of producers and consumers. All examples are taken from working Stampede$^{RT}$[104]/PTS example code slightly modified to be clearer in example form[1] –

[1]The modifications are all trivial (and all to the C++ code; Java examples are unmodified): in a few places, ampersands are omitted when passing structs as function arguments (the real API typically takes pointers to structs, but refs could be used in a more idiomatic C++ implementation) and changing certain type names from `struct` *structname* to the simpler `structname_t`.

PTS (Persistent Temporal Streams) is the name of the current implementation of the *temporal streams* programming model. Each example in this chapter is presented in somewhat idealized C++ and Java, with the C++ examples appearing on top and the Java examples appearing on bottom of each source listing. Two versions are presented to show both the procedural and object-oriented style interfaces to *temporal streams*. PTS has two parallel native implementations in C and Java. The C++ interface is not object-oriented because it is a wrapper around the C interface primarily providing default arguments and overloaded functions (which simplifies the API considerably).

**Channel binding:** The first example in Figure 6 shows how a programmer would initialize PTS and retrieve a channel descriptor to produce data or consume data. First the programmer connects to a particular instance of the PTS naming and registration system. Next the programmer calls an "add channel" function with a specified channel name. In its default form, the add channel function is actually an atomic "add or get channel" operation which creates the channel only if it does not exist and returns the channel information. Another (less frequently used) form of the "add" function exists which signals failure if the channel already exists, and a separate "get channel" function (which never adds the channel if it does not exist) is also available. After the default channel information is obtained, the programmer may create a dedicated data connection to the channel.[2]

**Simple Producer:** Figure 7 shows a simple data producer example. The example loops forever putting data items into the channel – each item is timestamped with the current time. Both the Java and C++ version are basically equivalent, but the C++ version shows

[2]In the first system implementation, the programmer can manage dedicated data connections and interfaces with channels via connection handles. In the second implementation, connections are pooled automatically by the system, so the programmer simply interfaces with channels using channel identifiers.

```
// Connect to front end and contact super-nodes.
rtsh = INIT_RT(front_end_uri);

// Add a channel.
chan = ADD_CHAN(rtsh, chan_name);

// Negotiate a dedicated data connection for bulk puts.
data_conn = CONN_CHAN(chan);

// Run the consumer function or producer function
...
```

---

```
// Connect to front end and contact super-nodes
RTSysHandle rtsh = new RTSysHandle(frontEndURI);

// Add a channel
Channel ch = rtsh.ADDCHANNEL(channelName,
                             rtsh.thisHost());

// Run the consumer function or producer function
...
```

**Figure 6:** Get a channel

the use of `conn_put` when providing an explicit item timestamp. The explicit timestamp is initialized to `now()`, which is also the default if no timestamp argument is provided to the put call.[3] In the Java example, the timestamp and data are associated with the `Item` object. When the `Item` is constructed, the timestamp may be explicitly specified; otherwise it defaults to the current time.

**Simple Consumer:** Figure 8 shows a simple consumer example where the consumer gets a single item at a time from the channel. The more general *get* operation retrieves all items falling within an interval (up to some limit, if specified), but the PTS API also provides separate convenience calls for getting a single item at a time.[4] In both examples, the consumer loops forever getting the newest item in the channel that has not yet been retrieved. When the consumer starts retrieving items, it starts with the absolute newest item in the channel.

The interval lower and upper bounds deserve further elaboration, as they are critical to the fundamental semantics of *temporal streams*. Remember that a *get* operation is defined on an interval: a standard *get* operation with a lower bound $l$ and upper bound $u$ returns all items contained within the interval $[l, u)$, starting with the first fully contained item. At first in our example, the lower bound is *newest* – the timestamp of the newest item in the channel at the time the call is made – and the upper bound is *now*. Notice that there are two "now" functions used in these examples: `v_now()` and `now()`. The former is the time variable representing *now*, which always evaluates to the current time whenever it is used, while the latter is simply a function that returns the current time when it was called. If you

[3] It is redundant to explicitly call `now` since it is already the default value when the timestamp is omitted, but it is done in this example so the timestamp value can be printed before the call.

[4] These convenience calls are just standard get operations with a limit of 1.

```
// Producer repeatedly puts items into the channel
void produce(conn_endpt_t *chan) {
  int rval;
  time32_t ts;

  while(true) {
    // timestamp -- 'now' is also the default if we
    // do not explicitly pass a timestamp
    ts = NOW();

    // print ts

    // Put the item into the channel
    rval = CONN_PUT(chan, data, data_len, ts);

    ...
  }
}
```

---

```
while(true) {
  Item item = new Item(dataBuffer);

  // Put the item into the channel with the
  // default timestamp of 'now'
  chan.PUTITEM(item);

  ...
}
```

**Figure 7:** Producer Example

save the result of `now()`, it stays forever fixed as a representation of the time when it was called (it is not a variable). `v_now()` stands for the time variable *now*.

After the first *get* operation is completed, we have an item with timestamp $t$. We reset the lower bound of the next *get* operation to be the $t + 1\mu$s – this is equivalent to the *next* time variable with a parameter of $t$ ($t + \epsilon$ where $\epsilon$ is the smallest timestamp granularity), but we show an explicit increment operation to demonstrate that it would be possible to add $2\mu$s or 1s or any value desired.[5] Thus the next *get* operation's lower bound is just past the item already retrieved and we will retrieve the next item in the channel we have not yet processed.[6] By default, the *get* operation will block until such an item becomes available. Non-blocking calls are also available in the PTS API (see Section 4.2 for an example). In this example, if the lower bound was instead initialized to *oldest*, then it would retrieve each live item in the channel starting with the oldest available when the consumer is started. That means it would quickly "catch up" by retrieving each live item in the channel one-by-one and then, once it has retrieved the newest item in the channel, the behavior will be identical to the consumer that starts with *newest* (i.e., it will retrieve each new item limited by the rate of the producer placing them into the channel).

## *4.2 Common Patterns using Time Variables*

Time-based retrieval of items from streams with time variables is a powerful pattern. It is also quite simple to perform commonly used, non-time based producer/consumer patterns with time variables. The consumer example in Figure 8 demonstrates retrieving all items in

[5]The Java version of the code calls the explicit `next` time variable operation. In PTS the timestamp granularity is currently $1\mu$s, but this could change in the future. For example, Java's JSR-166 "Concurrency Utilities" enhancements [119] (added in language release 1.5), provide a `System.nanoTime()` call in anticipation of the availability of – and demand for – nanosecond-granularity timing.

[6]Remember, this is because the operation starts with the first full item contained within the interval.

```
// Consumer repeatedly gets items from the channel
void consumer(conn_endpt_t *chan) {
  time32_t lower, upper;
  user_item_t item;

  // Initialize the lower bound to the virtual time
  // representing the newest item in the channel
  lower = NEWEST();

  // Initialize the upper bound to the virtual now time
  upper = V_NOW();

  while(true) {
    // Get one item
    item = CONN_GET_1(chan, lower, upper);

    // Do something with the item
    ...

    // Take the timestamp of the item and add one µsec,
    // so our next interval doesn't contain this item
    // This is equivalent to the ``next'' time variable
    lower = TIME_ADD(item.ts, ONE_USEC);
  }
}
```

---

```
Time32 lower = Time32.newest;
Time32 upper = Time32.v_now;

while(true) {
  Item item = chan.GETITEM(lower, upper);

  // Do something with the item
  ...

  lower = Time32.NEXT(item.getTimestamp());
}
```

**Figure 8:** Consumer Example

a channel sequentially starting with the newest item available at the time of the first call. In this example, the *get* operation is limited to retrieve a single item – specifically the first item matching the time interval provided. The lower timestamp bound is initialized to *newest* and the upper bound is initialized to *now* (the time variable now). After retrieving the first item *i*, `lower` is reassigned to *next*(*i*.ts), ensuring that the subsequent *get* call will return the item directly following *i*. If we do not limit the *get* operation to a single item, each call will return all of the items available in the channel directly following *i*.

If instead of modifying the interval lower bound, `lower` is kept as the *newest* variable for subsequent calls, we will instead retrieve the newest item in the channel, which could be the item following *i*, *i* itself (if nothing has been placed into the channel since) or some other item after *i* (if many items have been added since). This is rarely desired, however, because the same item *i* might be retrieved many times. To retrieve the newest item that has not yet been processed, we use the *newest-after* time variable. Figure 9 shows a Java example using *newest-after*. In this case, we will always retrieve the newest available item in the channel which has not been retrieved yet; this is different from the *next* example because using *newest-after* may skip items if several new items are added between get calls.

These two cases – `lower` = *next*(*i*.ts) and `lower` = *newest-after*(*i*.ts) – also demonstrate different strategies for handling overload conditions: in the *next* case, all new items are retrieved, while the *newest-after* case may drop items by skipping and never retrieving them if it falls behind. Instead of dropping items with the *newest-after* strategy, the application may choose to retrieve all items with *next* and then use some sort of content-based item dropping heuristic for selective processing.

Note that these semantics are also independent of whether the *get* operation is blocking

```
Time32 lower = Time32.newest;
Time32 upper = Time32.v_now;

while(true) {
  Item item = chan.GETITEM(lower, upper);

  // Do something with the item
  ...

  lower = Time32.NEWESTAFTER(item.getTimestamp());
}
```

**Figure 9:** *newest-after* Consumer Example

or non-blocking. As mentioned earlier, the examples provided assume blocking operations, but the code for non-blocking operations is quite similar. Figure 10 shows a non-blocking consumer example modified from Figure 9. The only major difference is the handling of the case where new items are not yet available: the application might choose to perform some sort of back-off, keep busy with other processing or perhaps request an older, skipped item to process while waiting for new data.

**Reference stream pattern:** As mentioned in Section 3.5, a client might want to perform a *get* operation from the same time interval on several related channels (i.e., synchronizing stream content). We refer to this pattern as the *reference stream pattern* and depict it graphically in Figure 3. In the presence of time variables, it is not sufficient to perform the same *get* operation on several channels in sequence (because the variables' meanings may change between channels). Instead, we must perform an initial *get* on a point-of-reference channel and then use the concrete upper and lower bounds from the actual operation to perform gets on subsequent channels. Figure 11 shows a basic example of how this might be

```
Time32 lower = Time32.newest;
Time32 upper = Time32.v_now;

while(true) {
  Item item = chan.getItemNB(lower, upper);

  if(item == null) {
    // Exponential back-off or:
    // Do other processing while waiting or:
    // Request an older item
    continue;
  }

  // Do something with the item
  ...

  lower = Time32.newestAfter(item.getTimestamp());
}
```

**Figure 10:** Non-blocking Consumer Example

```
GetResult gr = chan.GETITEMSWITHBOUNDS(lower, upper);

// new upper and lower timestamps based on
// instantiated variables
lower = gr.getConcreteLowerBound();
upper = gr.getConcreteUpperBound();
Items[] items1 = gr.GETITEMS();

Items[] items2 = chanConn2.GETITEMS(lower, upper);
Items[] items3 = chanConn3.GETITEMS(lower, upper);
...
```

**Figure 11:** Reference Pattern Example

implemented. The runtime already provides multi-channel *get* operations with a designated reference channel (as well as channel groups), so it is not necessary to actually write this code – the example is for illustrative purposes.

### *4.3 Overlapping Computation and Communication*

Consider a basic conceptualized feature detector depicted in Figure 12: the code simply takes a series of input items from `inputChan` and performs some per-item processing, producing a series of output data items into `outputChan`. Both `getItem` and `putItem` are used here with a straightforward synchronous interface. The way the code is written, however, processing (calling `processItem`) and data retrieval (`getItem`) are serialized. Fundamental design properties of the temporal streams architecture address this problem transparently (see Section 5.1).

Channels can be locally replicated (read-only, primary copy replication), which provides for transparent overlapping of computation and communication. With locally replicated channels, `getItem` does not have to wait for network communication. Items are

```
Time32 lower = Time32.newest;
Time32 upper = Time32.v_now;

while(true) {
  Item item = inputChan.GETITEM(lower, upper);

  // Perform processing on item
  data = processItem(item);

  // Produce output
  outputChan.PUTITEM(new Item(data, item.getTimestamp()));

  lower = Time32.NEXT(item.getTimestamp());
}
```

**Figure 12:** Unary Feature Detector Example

fetched to the local replica concurrently in the background. Figure 13 shows how a channel replica might be used in different ways. In one case, a programmer may explicitly create a local channel descriptor for use in one specific subroutine; alternately, one might choose to use the local replica globally (within the current peer) for all data retrieval on a particular channel. The key point is that the code in Figure 12 does not need to change to take advantage of overlapped computation and communication – it can maintain the simpler, straight-line synchronous structure.[7]

[7]We have only discussed overlapping communication with `getItem`; obviously `putItem` can also asynchronously transfer the item and return immediately, but that is trivial to implement and not due to any novel architectural property.

```
// Lookup a channel
Channel ch = rtsh.getChannel(channelName);

// Make local replica
Channel local = ch.localReplica();

// optional: Add a local replica to the
// local runtime's connection pooling
rtsh.addChannelConn(local);
```

**Figure 13:** Channel Replication Example

# CHAPTER V

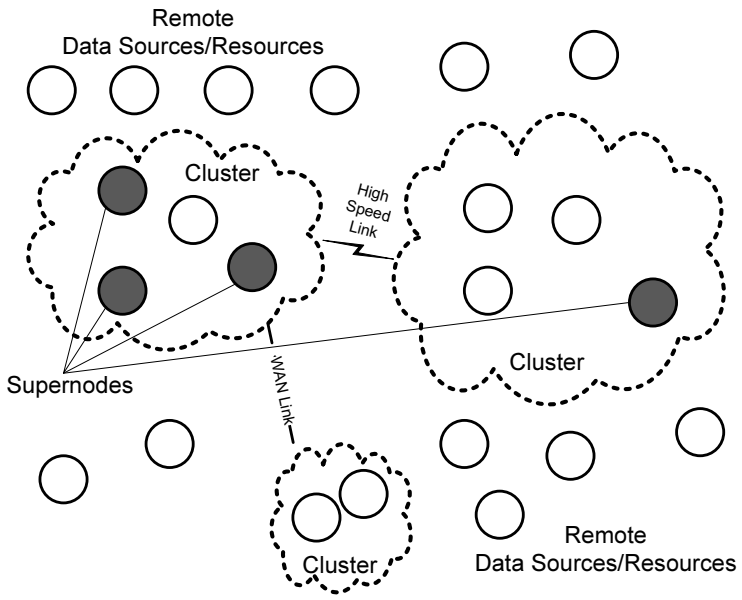
# ARCHITECTURE / SYSTEM STRUCTURE

The *temporal streams* programming model is an abstract way of viewing and expressing live stream analysis programs. In principle, this programming model could be realized in a variety of ways. The previous examples (in Chapter 4) show uses of this programming model from an API level, but most of the API is not specific to the underlying architecture or implementation. The system has been realized as distributed runtime providing the semantics of the abstract programming model (presented in Chapter 3). This chapter explores two major architectural facets: 1) the distributed communication architecture and 2) the storage architecture. In this chapter, note that "architecture" refers to the abstract system structure rather than a concrete software architecture. Several different realizations of this general system structure are presented in the following discussion of concrete software implementation (Chapter 6).

## *5.1 Distributed Communication Architecture*

Conceptually, the system is structured as a distributed runtime with peer-to-peer data transfer. One goal of this structure is providing scalability with decentralized communication and support for features like replication and multicast. The core of the system is a set of cooperating *peers* using the temporal streams runtime library – peers are data consumers or producers and host resources. In typical usage, a peer can be thought of as a multi-threaded process with a unique identity from the perspective of the system. The system also has a distributed, replicated directory storing system metadata (for instance, naming information

or the mapping between opaque temporal stream endpoints to network endpoints) which is accessed by peers. The nodes storing replicated metadata are called *supernodes*. Figure 14 shows an envisioned example configuration: the application spans several clusters and has external, WAN-connected data sources. A few nodes with high expected availability and connectivity are supernodes (storing system metadata).




Circles are nodes hosting peers; shaded circles are supernodes. The clouds represent clusters with fast internal networking, and the nodes not belonging to a cluster are external data sources, end-clients or other resources.

**Figure 14:** Conceptual Distributed System View

Almost all of the difficult concrete implementation decisions in the system revolve around hosting or accessing channels. Peers place timestamped items into channels (*put* operations) and retrieve items based on time intervals (*get* operations). How this is ultimately accomplished will determine the performance and scalability of the system. In the basic architecture, each channel has a canonical primary copy hosted at a single peer,

but they may be read-only replicated (primary copy replication) for capacity, availability, and also to enable overlapping of communication and computation without any changes to the application code. A channel's primary copy may also migrate dynamically to another peer if necessary. Architecturally, every peer is a first class entity which may host channels or interact with existing channels.[1] The system design should also support blocking and non-blocking data retrieval, push/pull style data-transfer, multicast, per-client transport negotiation and various other performance-related features.

## *5.2 Storage Architecture*

At a high level, the stream persistence interface is natural and intuitive; all an application needs to know is that data items are mapped to persistent forms using a known transformation and stored along with timestamp information. Underneath this abstraction, however, the data must be stored to "stable" storage somehow, and the potential design space is large. The streams could be stored to a local filesystem, a distributed filesystem, a DBMS, a distributed virtual block device, an object store (e.g., systems underneath Lustre [55] or Ceph [205]), or some other storage abstraction,[2] and there are many orthogonal design choices associated with each. In this section, we discuss several design properties.

### 5.2.1 Design Considerations

**Redundancy/Availability:** Some properties of the underlying storage mechanism manifest themselves as higher-level concerns. For example, an application may desire some form of redundancy so a stored stream does not become inaccessible due to disk or host

[1] Some peers, on low-end devices for instance, may run stripped down implementations without the ability to locally host resources. This is not a limit imposed architecturally, however.

[2] Boxwood's [132] persistent B-link tree abstraction is potentially quite well-suited assuming the interface supports "closest key" queries (i.e., finding the closest key to a value that may not be present in the B-tree)

failure. This could be accomplished in a variety of ways such as using a redundant, distributed storage mechanism as a backend, using primary copy replication, or making use of shared disks (e.g., via a SAN). [3]

**Free space management:** Another storage-level property exposed at a higher level is the management of free space. For high-bandwidth data streams, like video, an application will often want to use local storage as a *ring buffer* so the oldest stored data will be overwritten when storage is full. Support for some policies may already be provided by a storage backend, however. For example, the GPFS [177] distributed filesystem provides internal support for rich information lifecycle policies based on filesystem metadata – a policy could specify that old data can be reclaimed or moved to lower performance storage. Some databases also provide for similar policies, typically at a table (or some greater storage unit) level.

**External applications:** One may also want a persistent stream stored in a particular backend for reasons external to the application: for example, a user may want sensor readings inserted into a table in a relational database for offline analysis by another application or a third party.

**Suitability for workload:** The access patterns created by storing streaming data are atypical workloads for some potential backends. Stored items are never updated and are read rarely (relative to the number stored). From a storage perspective, the data is essentially append-only, which affords simple and efficient consistency management strategies. Ideally, the backend should not block concurrent reads of older data while appending newer

[3] Shared disks on a SAN with a dedicated network (like Fibre Channel) have the benefit of eliminating bandwidth contention between disk traffic and streaming data network traffic.

data. The system must also support ranged queries since data is accessed by specifying intervals (and the upper or lower bounds of a time range may fall between any stored item's timestamp). This property means that exact-match key/value stores are not directly suitable without the addition of an external range index.

When multiple streams are involved, the typical access patterns of storing many append-only streams simultaneously do not interact well with most general-purpose filesystem layouts [75]. Hyperion [75] addresses the problem of writing and querying multiple streams of captured high-data rate network traffic with a custom filesystem called StreamFS. The authors also present a "log file rotation" strategy for improving stream data layout on typical Unix filesystems; they also note that log-structured filesystems [173] are theoretically better suited than more conventional filesystems when applying this technique. In practice, however, their experiments found that SGI's XFS [188] outperformed NetBSD's LFS, despite the fact that LFS is log structured and XFS is not.

### 5.2.2 Pluggable Backends

To deal with diversity in requirements, we provide pluggable storage backends. Given the design tradeoffs discussed above, we support three backends: 1) a local filesystem backend (called `fs1`), 2) a distributed filesystem backend using GPFS (called `gpfs1`), and 3) a MySQL backend. Since we want to be able to handle multiple high bandwidth streams, we believe Hyperion's StreamFS [75] (or a slightly modified version) is best-suited to our target domain when using local disks. StreamFS is not publicly available,[4] so we have our own filesystem-based backend called `fs1` as a first-order approximation using the "log file rotation" approach presented in the Hyperion paper. We would also like to provide a

[4]We contacted the authors, but the system is a research prototype not yet in a suitable state for external release.

distributed storage solution with advanced ILM functionality, so we leverage the distributed filesystem GPFS for this purpose. A MySQL backend is provided for scenarios where streams need to be stored in a relational database (e.g., for analysis by other applications). In general, we do not believe that MySQL is a good general backend choice because it imposes a relatively large overhead on the storage process and was not designed for this particular workload (see Section 7.2 for related measurements).

Pluggable storage backends will allow the system to address diversity in application requirements. In addition, the architecture must support runtime-loaded data "pickling" code which allows the programmer to control how stream data is mapped to a persistent representation. The system should also be able to dynamically modulate the persistent representation to react to different kinds of resource contention (e.g., compressing data when the disk cannot handle the uncompressed data rate or choosing to forgo compression to lower the CPU load). As mentioned in Section 2.4, the ability to perform this kind of dynamic adaptation is directly facilitated by the temporal stream abstraction being high-level enough to capture certain domain-specific information: since the storage abstraction sees a sequence of timestamped data items rather than an opaque series of bytes, the system has higher-level information to exploit.

# CHAPTER VI

# IMPLEMENTATION / CONCRETE SOFTWARE ARCHITECTURE

This chapter details the software architecture and implementation of the previously described system structure (Chapter 5). The vast majority of implementation complexity is in the client runtime each peer uses, so we will describe it first and in the greatest detail. The system metadata components are more straightforward – the supernodes form a basic replicated key-value store, while the client library implements the particulars of the temporal stream programming model.

**Context:** The system has been implemented and revised incrementally in several iterations. All of the incarnations share the same basic system structure, but the concrete software architecture has evolved with experience. Since the architectural changes are informative, we will start by presenting the first complete design in detail and follow with a shorter summary of architectural evolutions in subsequent iterations (and a discussion of the tradeoffs involved). The first complete system has parallel interoperable native implementations in both C and Java, but here we will detail the C implementation as it is both more optimized and complete (and more involved).

**Roadmap:** This chapter covers a lot of ground, so following roadmap explains its organization.

- First Complete & Mature Architecture – First we will take a narrow view of the system and describe the inner workings of a channel hosted at a single peer in two parts:

1) the basic in-memory data management and communication hooks (Section 6.1) and 2) the storage/persistence stack (Section 6.2). In these sections we will take for granted the ability of peers to locate other peers. After the channel implementation is described, we will pull back and discuss the larger distributed structure of the system, describing how peers locate each other, bind to resources and communicate, as well as the handling of system metadata (Section 6.3).

- Discussion – After completing the description of the full system, we will follow with a discussion of the beneficial design decisions (Section 6.4).

- Subsequent Iterations – Following that, we will discuss changes in later iterations of the system in Section 6.5.

- Common Issues – Section 6.6 will then enumerate several key implementation issues at a higher level of detail – at the level of specific code, specific data structures, specific languages and libraries and various development pragmatics.

- Architectural Enhancements and Design Alternatives – Finally, in Section 6.7, we will discuss enhancements to the existing architecture and potential design alternatives.

## *6.1 Channels without data persistence*

Figure 15 shows the internal structure of a channel hosted at a single peer, including both in-memory data and communication hooks (this section, Section 6.1) and the storage persistent stack (Section 6.2). In this section, we will describe the basic in-memory data and communication hooks – the left side of the architectural diagram.

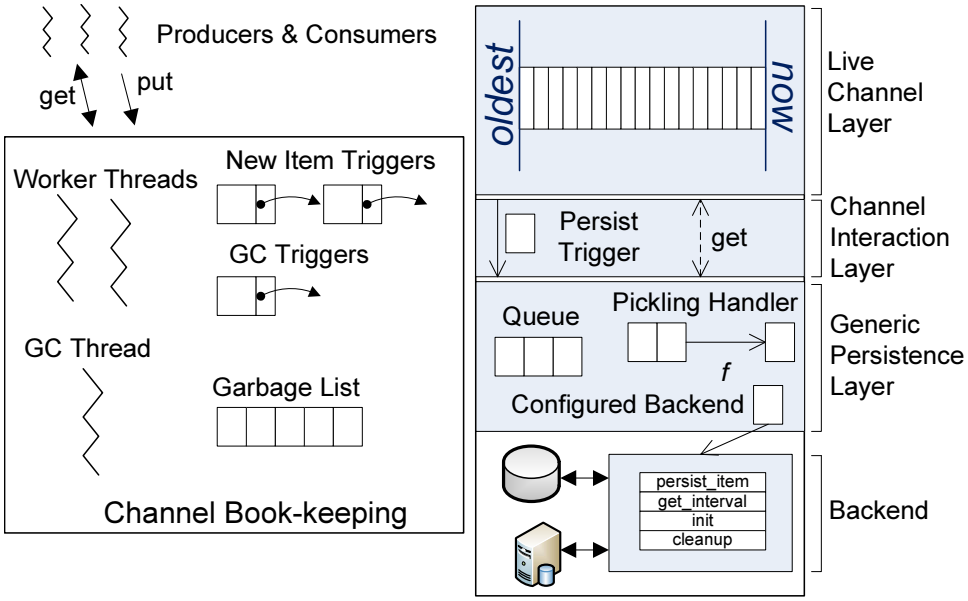



**Figure 15:** Peer Channel Architecture

**Basics:** Broadly, a channel is just an indexed data structure holding timestamped items and accompanying communication interfaces for peers to store and retrieve items. A channel stores current live stream items ordered by timestamp; items older than a given *currency* bound (e.g., 30 seconds) are automatically reclaimed. Conceptually, a channel may be viewed as an ordered list of data items and associated metadata (e.g., timestamps) located at the peer hosting a channel. Each peer has a single *gatekeeper* TCP/IP endpoint where other peers can either interact with any channels hosted locally or negotiate a separate dedicated connection for bulk data transfer. The transport protocol of dedicated connections can be negotiated on a per-connection basis (e.g., shared memory for collocated processes, RDMA or SCTP [157] within a cluster, etc.). A pool of worker threads is used to handle remote *get*/*put* requests on dedicated connections.

**Connection Management:** When performing *get* or *put* operations, a channel is identified by a *channel descriptor*, which is an opaque reference to a particular channel data

connection. Each peer has a table mapping channel descriptors to concrete connection endpoint information, which acts like a cache: normally, channel operations use the cached information and no metadata lookup or binding is necessary. When a channel moves or a new connection to a channel is needed, the runtime contacts the system metadata directory to find out which peer is hosting a channel and then contacts the peer's gatekeeper endpoint to negotiate a data connection. For more information on how peers locate and contact each other, see Section 6.3.

**Data Movement:** Item data and metadata are never copied within the system and stream data transfer via networking always uses scatter/gather I/O. For *get* requests involving many items, the first ten item lengths are included in the initial request header so the receiving client can use scattering I/O into preallocated buffers (items after the first ten are batched in chunks of 32). Additionally, since stream data items are immutable, item metadata is pre-serialized into a 20-byte item header placed before each item's data when items are initially put into a channel. See Section 6.6.2 for more detailed information on the wire protocol.

**Triggers:** One of the main concerns of a channel's internal bookkeeping is not just keeping track of items in a data structure, but item lifecycle issues – i.e., actions taken when items are newly added and when they are reclaimed, as well as related storage management. Many related functions are managed with a flexible *trigger* mechanism. A channel's integral trigger mechanism provides two different types of triggers: 1) *garbage collection triggers* and 2) *new item triggers*. Both types of triggers are functions that apply to a single item at a time. Garbage collection triggers are invoked when an item is about to be removed from the channel's "live" data and either freed or placed on a garbage list; new item triggers

are invoked when a new item is added to a channel. While this is a very simple concept, it is also remarkably flexible. Triggers are used to implement a variety of functionality – new item triggers are the basis for replication of channels, multicasting channel data, an optional push-style programming interface, and channel groups. For example, to set up a copy of channel A replicated at host B, the system creates a new locally hosted channel at host B, and sets up a new item trigger on channel A to send each item to the replica. Any host can now use the copy by updating its channel descriptor table to point to the replicated channel.

Using a locally replicated channel in the manner facilitates overlapping communication and computation without any special effort on the part of the application programmer. Consider a simple content transformation loop: a client gets an item from a channel, runs some content analysis or transformation function on it, and places the results in another channel. If the channel is remotely hosted, and the programmer performs standard blocking gets on a channel, the communication and content analysis will be serialized. With a local channel replica, however, the same client actions allow the overlapping of communication and computation because the get operations are all performed locally. Data transfer from the remote channel happens in the background concurrently to populate the local channel replica with data items.

**Trigger Execution:** To execute triggers, each channel maintains a list of functions to call for each trigger type and invokes them sequentially and in the execution context of the thread that added an item or caused an old item to be prepared for garbage collection. Consequently, trigger functions are expected to have short, bounded execution times. When a trigger is added, an initialization function is run which can set up an event queue and a dedicated listening thread or bind to a shared thread/thread pool for asynchronously servicing

longer triggers (analogous to "bottom half"/second-level processing for interrupt handlers). Triggers can be loaded by name statically or dynamically (via `dlopen`).

**Heap Management:** The C-based runtime uses reference counting for internal storage management of channel data. Since C does not have automatic storage management and the runtime is multithreaded, the refcounts ensure that an item is not freed while still in use (for instance, while sitting on an event queue or while data is waiting to be sent over the wire). Refcounts are maintained with compare-and-swap atomic builtins provided by gcc (since 4.1) or mutexes when atomic operations are unavailable.

**Channel Garbage Collection:** Without persistence, channel garbage collection is easy: since a *put* call places a single timestamped item into a channel, we just check to see if we can reclaim the oldest item in the channel after a *put* call. If the span between the newest and oldest items is greater than the channel's specified currency bound, the item is removed from the live channel and the system invokes the GC trigger functions. The last trigger will either place the item on a garbage list if its refcount is non-zero or immediately free it otherwise. If the item isn't immediately freed, we walk through a small fixed number of items on the garbage list and free those with refcounts of zero. There is no need for a background GC thread because new garbage is only generated when old items are displaced by newly arriving data, so the system can maintain stasis by doing a small amount of GC work cooperatively during each *put* call.[1]

[1] Since this strategy only reclaims older items when new items are placed into a channel, some old items may remain in a channel longer than strictly necessary. As described here, the programming model does not make any strict guarantees that items older than the garbage collection bound will *not* be available, merely that they can be collected.

**Monitoring:** Each channel can also dynamically monitor and control memory and network usage by collecting statistics on the total size of live data in a channel, as well as the number of bytes transferred in *get*/*put* operations. For example, the system could lower the number of seconds of live data kept in a channel in response to memory pressure, perhaps within some specified upper and lower bounds provided by the application programmer.

**Push/Pull:** Since the programming model specifies operations in terms of a familiar data structure, the default transfer semantics for data items are in terms of *get* and *put* operations for consumers and producers, respectively. This naturally leads to the default item transfer semantics of a *push* model for producers and *pull* model for consumers.

Realistically, there are two orthogonal dimensions of push/pull variance: 1) the programming interface and 2) the actual data transfer paradigm. For the programming interface, the pull model for consumers would imply explicit *get* operations, while the push model would cause a callback to be executed for each item of interest. For producers, the push model implies explicit *put* operations, while the pull model would use a periodic sampling of a predefined data area or periodic callbacks. The data transfer paradigm affects how remote data items are actually transferred: a push model implies that items of interest are sent when ready, while a pull model implies that items are fetched on demand.

For channels, consumers may utilize a *push* programming interface by registering a trigger to be invoked when a new item is available on a local channel.[2] Consumers may utilize a *push* data transfer model by registering a new item trigger on a given channel.

[2]One can also provide a *pull* programming interface for producers. As a concrete example, consider a data producer where a video camera is attached via a video capture card and the capture card memory maps the current frame into a specified memory region. In this case, the pull callback would periodically sample the region and put the data into a predetermined channel. This paradigm is convenient, but it is only directly applicable to a subset of data producers.

When a new item is added to that channel, the trigger will execute and send the item to the interested host or hosts. A local channel replica is created on the interested host to receive items from the trigger action. Local actions utilize the channel copy as a data source. Again, this is a simple implementation strategy, but it shows the versatility of the trigger concept.

**Channel Groups:** Channel groups with a concrete reference stream can be implemented by broadcasting new item timestamps to the other streams in the group (using a new item trigger). When a group is synchronized to *oldest*, new item timestamps are broadcast by all streams. Although this strategy is simplistic, it is typically not a bottleneck for several reasons: in most of the common anticipated use-cases for channel groups, the channels will be hosted on the same peer or peers on the same local network. This is due to the fact that channels in a group are typically closely related (e.g., part of a coherent multi-modal stream broken into components, such as video/audio or two video cameras for stereo vision). In these cases, efficient shared-memory or multicast communication can be utilized. The bandwidth for broadcasting timestamps is miniscule compared to the bandwidth required for most media streams, and the latency between hosts already provides a lower bound on the synchronization accuracy. Larger or more widely distributed channel groups may require more advanced techniques.

## *6.2 Channel Persistence*

At a high level, the persistence interface for channels seamlessly integrates storage into the live stream abstraction. As mentioned earlier, our persistence architecture provides pluggable storage backends for flexibility. Our concrete architecture for the persistence mechanism is separated into three general layers: 1) the *channel interaction layer*, 2) the

*generic persistence layer* and 3) specific persistence *backends*. These components are depicted on the right side of Figure 15. The persistence backends are loaded dynamically and handle interfacing with a particular storage mechanism (e.g., a filesystem or object-store or database). Both the generic persistence layer and concrete persistence backends provide a simple API with four basic calls: `persist_item` and `get_interval` as well as `init` and `cleanup`. `persist_item` and `get_interval` directly correspond to the live channel *get*/*put* operations.

### 6.2.1 Channel interaction layer

The channel interaction layer is the small set of hooks in the existing channel implementation which interfaces with the generic persistence layer. For channel *get* operations, this consists of the logic to interpret get types (`ANY`, `LIVE`, etc.) and to call down to the persistence layer if stored items will be needed. If a *get* operation is performed on interval $[l, h)$ and some live item has a timestamp $\leq l$, then no call to the persistence layer is needed. After a `get_interval` call to the persistence layer is made, the channel interaction layer also handles placing temporary items retrieved from the storage backend on the garbage list.

Triggers are used to send items to the lower levels of the storage stack by calling `persist_item` in the generic persistence layer. If lazy persistence is requested, a GC trigger will be used; eager persistence uses a new item trigger. The channel interaction layer also contains routines to initialize the persistence interface. When a channel is initially marked as persistent, a background garbage collection thread is spawned since *get* operations spanning persisted items may create significant additional garbage and our previous strategy may not be able to keep up (particularly if *put* calls are rare).

### 6.2.2 Generic Persistence Layer

The generic persistence layer sits between the live channel and a particular concrete storage backend. It maintains a small set of items to be persisted in batches, and is responsible for calling pickling handlers to map some number of items into a persistent representation. The `persist_item` call simply places an item on a processing work queue to be handled asynchronously by a dedicated worker thread. This structure serves several purposes: 1) it prevents the `persist_item` call from blocking long (since it is called from a trigger), 2) queueing is necessary to support pickling handlers that transform N items to 1 item (we must batch N items), 3) if eager persistence is used on a channel with multiple producers, some queueing is necessary to ensure items are written out in temporal order,[3] and 4) it allows the generic persistence layer to serialize writes to the backend.

Several of these properties simplify the assumptions a storage backend must deal with. For example, serializing writes to the backend by the persistence layer simplifies backend implementation – it may assume there are no concurrent writers, although a single writer may overlap with item reads. Another feature of the generic persistence layer is that it guarantees that items will be presented in temporal order to the storage backend, which again can simplify the backend's implementation.

To process a `get_inverval` request, the generic persistence layer must search its work queue for items that are waiting to be persisted as well as call down into the concrete storage backend layer to retrieve items that have reached "stable" storage. Finally, the generic persistence layer is also responsible for dynamically loading storage backends and pickling functions when a channel is first marked as persistent.

[3]The queuing time could be limited to the potential clock skew between producers plus an estimate of propagation time.

The generic persistence layer can also monitor and react to different kinds of resource contention: by measuring the latency of backend `persist_item` calls, it can determine if storage contention is too high. Similarly, by timing pickling handler execution, it can estimate CPU load. The generic persistence layer can adjust to these conditions by switching between pickling handlers or disabling pickling. The persistence layer primarily affects CPU and storage contention; network and memory usage can be monitored and controlled by the live channel implementation.

### 6.2.3 Storage Backends

The architecture provides pluggable storage backends to deal with a wide variety of application requirements. As mentioned earlier, an application scenario may need sensor streams sent to a relational database for future analysis by third party applications. The particular storage backend used is dynamically selectable on a per-channel basis at the time the channel is made persistent.

Each storage backend has a fairly simple data model imposed on it by the generic persistence layer: channel items are immutable and timestamped items arrive in order from a single writer thread. Reads may occur concurrently and from any number of reading threads. A storage backend is responsible for storing items and retrieving items by timestamp. The retrieval interface (`get_interval`) is identical to a live channel *get* operation, so the backend must have some way of performing range queries that return all items covering an interval $[l, h)$ (and the lower and upper bounds of the interval may fall between timestamps, so it must support "nearest key" queries).

As a concrete example, consider a conceptual relational database backend (as it is the most straightforward for illustrative purposes). `persist_item` would be implemented as

depicted in Figure 16 – the ? values represent the concrete timestamp and data values. get_interval's implementation is shown in Figure 17, and the ? values represent the lower and upper timestamp bounds.

```
INSERT INTO table(tstamp, data) VALUES(?, ?)
```

**Figure 16:** persist_item implementation

```
SELECT (tstamp, data) FROM table
    WHERE (tstamp >= ? AND tstamp < ?)
```

**Figure 17:** get_interval implementation

Obviously the concrete implementation would need to interface with the database library and perform many other tasks, but the basic core of the storage backend is conceptually simple – the storage backends are only responsible for implementing persist_item and get_interval calls (as well as initialization and clean-up routines). The following subheadings describe the three concrete backend implementations:

**MySQL backend:** The MySQL backend is not designed for streams with high data rates, but it is certainly appropriate for low bandwidth sensors. persist_item just inserts a tuple with (timestamp, data) into a specified table and get_interval performs a SELECT of items with timestamps in the interval [low, high). The timestamp column should have an index suitable for range queries (e.g., B-trees) for efficient execution of the SELECT query. Currently the backend stores item data as BLOBs, which is not very flexible. A more advanced implementation could provide a richer interface by allowing user-defined functions to map binary item data into some number of separate data items matching the desired schema.

Due to the design of the MySQL's thread-safe C client library (libmysqlclient_r),

the MySQL backend uses a fixed set of worker threads and serializes backend *get*/*put* requests.[4] *Put* requests are executed asynchronously, while *get* requests naturally block waiting for items.

**Filesystem backend:** The `fs1` filesystem backend is implemented as a lightweight overlay on top of SGI's XFS [188], but it could also be implemented directly on a block device. Our implementation can run on any Unix filesystem and doesn't rely on features particular to XFS (such as guaranteed rate I/O or realtime subvolumes); we use XFS because it generally has the best overall performance for these workloads.[5] In fact, media serving and streaming video are the motivating applications mentioned by XFS's designers. The data layout of our `fs1` backend is quite simple and uses the properties provided by the generic storage layer to avoid unnecessary complexity and synchronization. We use the log file rotation approach presented in the Hyperion paper [75].

A backend needs to store timestamped data items in order and retrieve them by bounded time intervals. To accomplish this, `fs1` uses a two-level indexing scheme. A given channel has a single top-level index file and many individual data files, each with a second-level index; a data file's size is roughly bounded by a *chunk size* parameter (default 16MiB per file) and the small, fixed index is stored at the beginning. The top level index file contains an array of 64-bit timestamps corresponding to the lowest timestamp present in a given data file. A data file's index is simply an array of (64-bit timestamp, 32-bit item data offset, 32-bit item length) entries. Figure 18 depicts this index structure. A new data file is created when the current file's size exceeds the chunk size bound or when the index is full. Indices

[4] By default there is a single dedicated get thread and put thread.

[5] Next-generation filesystems like ext4 [136, 116], btfs [4] or HAMMER [77] were not mature enough at the time of the experiments to evaluate.

are preinitialized to a sentinel value (the timestamp that represents infinity), and a binary search is used to find the lower bound for `get_interval` in both the top level index and the index for the data files. This structure is very similar to the ISAM (Indexed Sequential Access Method) [90] data indexing format.

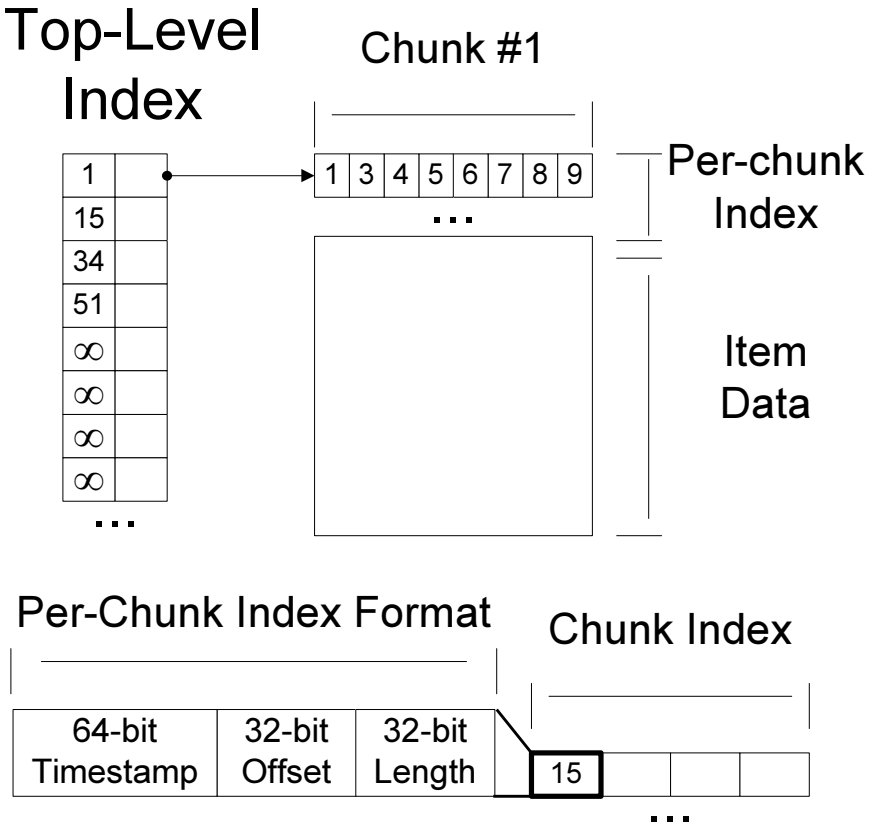



**Figure 18:** `fs1` Index Structure

Since the generic persistence layer guarantees that items arrive in order and there is only a single writer, the data files are append-only, which leads to simple logic for `put_item`. Items are added by first writing file data, adding the offset and length to the index and finally by writing the timestamp into the index. This allows readers to co-exist with writers without much synchronization – a memory fence may be needed to ensure that the offset, length

and file data appear before the timestamp write becomes visible to readers, depending on underlying hardware write ordering semantics.

**Distributed filesystem backend:** This backend is a variant of our non-distributed filesystem backend. It stores streams as whole files with a separate multi-level index directly on GPFS [177], which is already relatively well-tuned for streaming workloads. This backend also takes into account desired replication/failure semantics in placing data into proper filesets/storage pools with GPFS tools.

### 6.2.4 Lazy Versus Eager Persistence Tradeoffs

The choice of a channel's persistence policy changes certain corner cases of stored data retrieval; both policies imply tradeoffs in implementation complexity of different components. Even though our implementation supports both, certain simplifying assumptions are enabled by the choice of a particular policy. To review, lazy persistence stores items as they are expired from the live channel data while eager persistence stores items as they are initially added to the channel.

Conceptually, lazy persistence is appealing because it provides a nice ordering property for items: stored data is always older than data queued in the generic persistence layer, which is in turn older than "live" data. Despite this property, the *get* procedure must always adjust timestamps of the get interval as each layer of the hierarchy is traversed to exclude already considered items, because otherwise an item could be considered multiple times if it happens to move between layers during an in-progress *get* operation. Lazy persistence does simplify the generic persistence layer – it can place items in a simple FIFO queue, since items are guaranteed to arrive in order (after being ordered in the live channel data structure). Eager persistence instead requires that the generic persistence layer re-order

items, at least when multiple producers may place items into a given channel.

On the other hand, eager persistence avoids the need for synchronization in the generic persistence layer and in storage backends to avoid the potential for temporarily "lost" items. For example, consider a basic persistent channel with no pickling transformation function (i.e., each item is stored as is). An item that is just dequeued by the generic persistence layer will be pushed down to a backend, which is responsible for storing the item. When using lazy persistence, at some point before the item has reached stable storage in the backend, it will not be visible in the live channel data, the generic persistence layer's queue or the backend's data. This problem is also mirrored in any storage backend that uses a similar queueing mechanism for *put* operations. Solving this problem requires additional synchronization, which entails related overhead. Eager persistence provides overlap between live and stored items, so items will have reached stable storage and be visible by the time they are removed from a channel's live data.[6]

Finally, although eager persistence avoids temporarily "lost" items, there is a related problem with data duplication when N-to-1 pickling functions are involved. Consider a pickling function that maps five items to a single output item. When a group of five items initially arrive in a channel, a combined output item will be created and stored. Later, some number of these items will be garbage collected, and it is possible that the five items will be split by having some collected as garbage and some remain in the channel's live data. For a brief window until the rest of the original five are removed from the live channel, a *get* operation may see a persistent item created by transforming the original five items as well as some remainder of the original five items left in the live channel data. This problem

[6]This property may still be violated in an overload situation if items are queued up longer than the garbage collection time for a given channel; we can account for this situation the same way we do with lazy persistence, but such a backlog of items would likely imply the workload is too high for available resources.

can be avoided by noting the timestamp of the first item persisted in a channel and always removing items from live channel in batches of N (where N is the "arity" of the pickling function) starting from the first persistent item.

## *6.3 Distributed Structure*

So far we have described core channel functionality specific to peers – peers are data consumers or producers and host resources. In this section, we will discuss the general distributed structure of the system. This includes how peers locate other peers and how mappings between opaque channel endpoints and peers/underlying transport endpoints are managed. The system also includes *supernodes* and a *front-end*. The supernodes form a distributed metadata directory, and the front-end is a well-known location storing a list of supernodes (used to bootstrap new peers and supernodes joining the system).

### 6.3.1 Entity Metadata & Endpoints

As background for the following sections, here we will introduce the various system-wide metadata. Each peer is uniquely identified by a global *host identifier*. In our implementation, these identifiers are simply incrementing integers, but they can be any opaque identifier (UUIDs [20], for example). The system keeps a map linking host identifiers with actual communications endpoints (i.e., an IP address and port number). Peers can also be named, and their names are mapped to host identifiers, so finding a concrete communication endpoint from a name takes two lookups.[7]

Like hosts, each channel also has a unique global opaque *channel identifier*. For each channel identifier, the system maintains a mapping to a single host identifier. This mapping

[7] *name* → *host id* → *endpoint*

names the current peer with a canonical copy of the channel. Although channels may be read-only replicated, one peer is responsible for the canonical, primary copy at a given time. Channels can also have names – this is accomplished by mapping names to channel identifiers.

### 6.3.2 Peer Interactions

A peer is just a single distinguished participant in a distributed application using *temporal streams*. Architecturally, there is no required mapping between peers and threads or processes – in typical implementations, a peer will be a single multi-threaded process. As mentioned earlier, peers are identified by opaque unique identifiers and can also be assigned unique names. Peer names are useful in creating channel endpoints hosted by a particular peer (they are also used for mailboxes). Peers can join and depart the communication dynamically, although peers hosting resources should negotiate the migration or shutting down of data sources in use.[8] As mentioned earlier, each peer has a canonical initial point of contact in the form of a TCP/IP endpoint called the *gatekeeper*.

Each peer has several local caches to store mappings between opaque host/channel identifiers and transport endpoints. In the common case, channel operations (getting or putting an item, for example) will simply require a table lookup for a cached endpoint descriptor. If the cache lookup fails or the endpoint is found to be invalid, the system will attempt to resolve the missing connection endpoint by contacting an appropriate entity.[9] The major cached entities are *host gatekeeper endpoints*, *channel data endpoints* and *channel identifier to host identifier mappings*. A cache miss on a channel data endpoint will cause the system to contact the host gatekeeper for channel information – this host is the

[8] Ungraceful departure is possible, but more expensive to the remaining participants (due to timeouts, etc.).
[9] The peer cache functionality is similar in spirit to ARP caches.

last peer known to be hosting the channel in question. A cache miss on the host gatekeeper endpoint table or the channel identifier to host identifier mapping will cause the system to contact a supernode for appropriate information. It is also possible the cached mappings are stale; in this case, the operation will fail and will be treated as a cache miss.

For example, consider peer *A*'s use of a stale channel to host identifier mapping (when a channel has moved to another peer): first, peer *A* contacts an old peer previously hosting the channel and asks to establish a persistent connection to the channel. The reply indicates that the channel is no longer hosted there, and the requester will treat this as a cache miss and ask the metadata directory for an up-to-date mapping.

The cached transport endpoint descriptors can also include persistent connections rather than just passive port/host endpoint information. Since connection establishment is usually expensive, producers and consumers will open persistent data connections to channels for repeated use. The management of such persistent connections has changed over time and between implementations. The initial prototype uses explicit user management of persistent connections – a channel identifier object refers to a non-persistent default connection, and programmers explicitly create dedicated persistent connections and use them in lieu of the default connection. In a later revision, the table containing cached channel data endpoints was extended and indexed by a pair consisting of the opaque channel identifier and a locally unique thread identifier. This modification allows multiple threads in a single peer to have individual persistent connections to the same channel, while still referring to channels through generic channel identifiers. When a specific thread does not have a unique table entry, the default connection (or connection pool) is used. In both cases, persistent connection re-establishment[10] can be managed by maintaining metadata indicating the lineage of

[10] in case channels move or persistent connections time-out

a connection (e.g., an entry's preferred transport protocol and parent channel). These two approaches both have advantages and downsides; manual management is more flexible, allowing the programmer fine-grained control over the use of persistent connections, but this flexibility adds complexity to user-visible code. A later evolution of the system moved to a peer-wide, managed connection-pooling approach (see Section 6.5).

### 6.3.3 Supernodes

Supernodes form a simple distributed directory. Our definition of a supernode is similar to that in several P2P systems like FastTrack and Gnutella [208]. Any node in the system can "volunteer" to be a supernode, but the envisioned configuration has a small number of supernodes hosted by peers with high availability and connectivity. Peers will preferentially order their list of supernodes to contact by latency, so it is beneficial to place supernodes within various network segments to provide low-latency operation to local peers.

Supernodes store several directories of interrelated information: 1) a directory mapping *opaque host identifiers to gatekeeper endpoints*, 2) a directory mapping *opaque channel identifiers to host identifiers*, 3) a directory mapping *channel names to opaque channel identifiers*; and 4) a directory mapping *host names to opaque host identifiers*. Peer queries to supernodes are straightforward operations, and updates cause propagation to all other supernodes. Supernodes are kept synchronized using standard consensus techniques. Although the updates require agreement among the supernodes and therefore the cost grows with the number of supernodes, directory update operations are relatively infrequent and are not "critical path" operations. In general, the system uses a query and retry approach to metadata lookups (i.e., from peers to supernodes), which simplifies the caching system considerably. Peers keep locally-cached metadata which may be stale, but incorrect

stale mappings will either never be re-used[11] or they will cause failures leading to retried lookups.

Like peers, supernodes can join and leave the system at will. However, since supernodes are typically hosted on high-availability nodes and use minimal system resources, we expect that supernodes will rarely leave the system voluntarily. Since supernodes are "volunteers," there is currently no election process/handoff mechanism if no supernodes remain. Although the peers and supernodes are conceptually distinct components, one possibility to mitigate the potential problem of supernode loss is by adding supernode functionality to the peer library – in that case, all running peers have the capability to serve as supernodes and the system can automatically monitor the level of remaining supernodes and "deputize" peers as the population shrinks. The original supernode implementation takes this approach, but later revisions of the system omit it, favoring a simpler supernode implementation and the decoupling of the peer library from supernode hosting functionality (see Section 6.5).

### 6.3.4 Front-end

The front-end is a very simple component that provides a known location to allow peers and supernodes to join the system. The front-end's primary responsibility is keeping the list of supernodes in the system. In its most basic form, this could be maintained in simple DNS records or using other directory services like LDAP. A more advanced implementation can also perform heartbeat probes to supernodes and monitor liveness/reachability. This functionality might be redundant, however, depending on the particulars of supernode

[11] and potentially evicted for capacity

implementation – since supernodes use distributed consensus techniques, they will typically also have failure detectors.

### 6.3.5 Example Interaction

In order to concretize the previous discussion of the distributed communication structure, consider a simple running instance of an application using *temporal streams*. For illustrative purposes, we will use the basic interaction presented in Figure 19. This example code performs the following actions: 1) binds to the front-end and supernodes (i.e., joins the system), 2) looks up channel information, 3) creates a dedicated data connection to a channel, 4) retrieves an item from a channel, and 5) puts a transformed data item into the same channel.

Here we will break down the internal communication for each step (see Figure 20):

1. `connect` – The runtime contacts the front-end and gets a list of supernodes.

2. `addChannel` – The runtime contacts a supernode to perform an add or lookup on the channel with the specified name. If the channel does not exist, it is created; otherwise the existing channel information is returned. After the information is returned, the peer hosting the requested channel is known; consequently, the runtime updates the local cache to reflect the channel identifier to host identifier mapping. The runtime then contacts the channel host to verify the metadata is correct. In the process of contacting the channel host, the runtime updates the local cache to provide current value for the host's gatekeeper endpoint.

3. `persist` – The runtime needs to perform a channel operation, so it looks up the channel's host in its internal cache. To contact the host, it looks up the host identifier

```
// Connect to front end and contact super-nodes
PTSSysHandle ptsh = PTSSysHandle.CONNECT(frontEndURI);

// Get a channel
Channel ch = ptsh.ADDCHANNEL(channelName,
                             ptsh.thisHost());

// Get a dedicated connection
ChannelConnection chConn = ch.PERSIST();

// Get an item
Time32 lower = Time32.newest, upper = Time32.v_now;
Item item = chConn.GETITEM(lower, upper);

// Perform processing on item
data = processItem(item);

// Produce output
chConn.PUTITEM(new Item(data, item.getTimestamp()));

...
```

**Figure 19:** Example Interaction Code

to gatekeeper IP mapping in its local cache. It then contacts the channel's host peer and requests a dedicated connection to the given channel.

4. `getItem` – To service this channel operation, the runtime uses the dedicated connection to send a *get* request and retrieve the results.

5. `putItem` – The runtime uses the dedicated connection to send a *put* request and item data.

If a persistent connection is broken, the system will simply try to re-establish it using last known locations for channel data. Similarly, if any cached information is incorrect,

the runtime will treat the problem as a cache miss and re-query an appropriate data source (peers or supernodes).

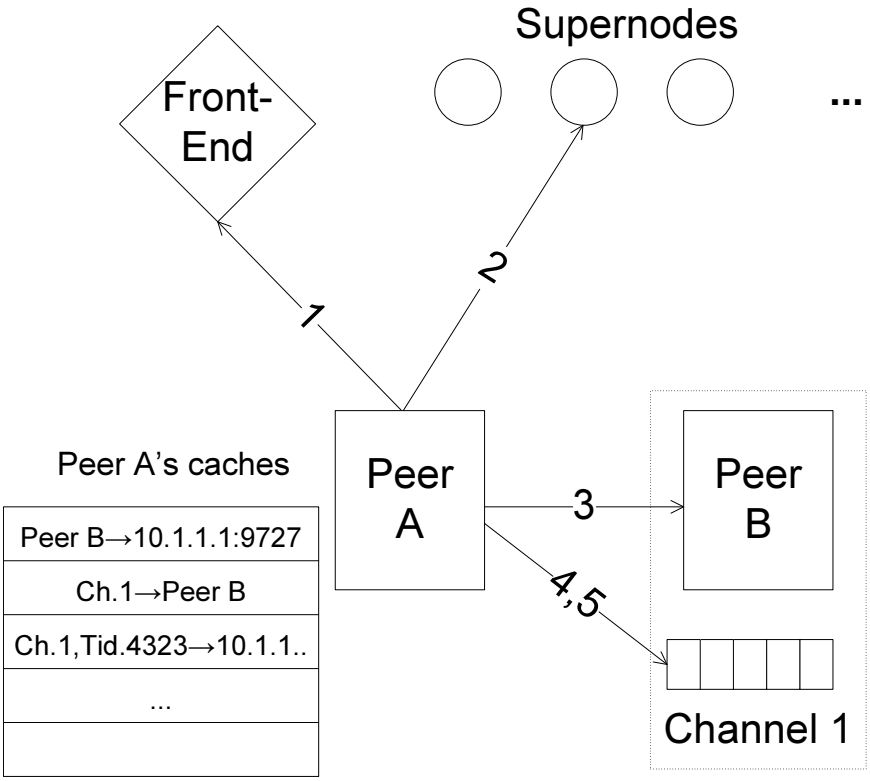



**Figure 20:** Example Interaction Depiction

## *6.4 Discussion of Design Considerations*

Many of the programming model and system architectural features are explicitly designed to provide key beneficial properties (e.g., flexibility, efficiency in implementation, etc.). Some of these properties are mentioned in Section 2.4; in this section, we will discuss the system-level implications of various design properties.

**Internal Runtime Structuring:** Careful design of the runtime internals yields a surprising level of flexibility from the channel primitive – for example, judicious use of layering

provides pluggable storage backends for free while also vastly simplifying the implementation complexity of a backend. The basic channel implementation provides an integral *trigger* mechanism which is simple but very versatile. Triggers are used to implement a variety of functionality; they are the basis for replication of channels, multicasting channel data, an optional push-style programming interface and the implementation of channel groups. The available trigger points, new item triggers and garbage collection triggers, reflect two key points in the life-cycle of channel data items: 1) when new items are added and 2) when old items have outlived the window of current data and "expire." Providing hooks at just these two points provides a straightforward way to implement a wide range of functionality that fits naturally with the framework set by the programming model.

**Distributed Structure:** The REST (Representational State Transfer) architectural style is a set of conceptual network architectural design principles guiding the design of the World Wide Web [85] (and generally applicable to hypermedia systems). The Web is one example of a system primarily following REST's principles, and those principles are often cited as reasons that the Web can scale to unprecedented levels. Many distributed systems can benefit from understanding REST's design principles. Several of REST's desired properties come from the related "Layered-Client-Cache-Stateless-Server (LC$SS)" protocol interaction style, which takes a basic stateless client-server model augmented with the ability to add inline caches, proxies and other intermediary components [85]. Although *temporal streams* is not a distributed hypermedia system, it nonetheless uses and benefits from many of these architectural principles.

The system presents a simple standard get/put interface between all components which refer to specific resources – the resources are data items in a channel, and a time interval plus a channel identifier name these resources. The system has a client-server architecture

explicitly avoiding client state at servers,[12] which allows caching and intermediaries (read-only replicated channels are just one example) – these features are used to enhance scalability and performance. Volatile client state, like persistent connections and data caches, is often used for efficiency, but such state can be lost and regenerated, only affecting performance; the ability to drop state without affecting correctness provides enhanced flexibility and robustness and greatly simplifies many consistency issues.

In addition, the system exploits *eventual consistency* [201] to avoid complex and expensive mechanisms for maintaining metadata consistency. Components interacting with system metadata (and volatile state, like persistent connections) are built to retry lookups (see Section 6.3.3). Although certain metadata updates require conceptually atomic consensus among metadata directory components to avoid "split brain" issues (e.g., two peers thinking they hold the canonical copy of a given channel), channels can move between peers without the need to notify current channel users. Since peer and channel endpoints are opaquely identified, they can move; system components are designed to re-establish broken connections and revalidate cached metadata.

**Programming Model:** High-level features of the programming model itself can significantly influence lower-level efficiencies of concrete implementations by imposing certain semantics or enabling various optimizations. One particular example is the design of time variables – time variables like *newest-after* and *next* are designed to be predicated on explicitly provided timestamps (see Section 3.4). Conceptually, this predication pushes explicit

[12]Note that this use of "client" and "server" terminology does not imply a traditional client-server architecture with statically determined roles. It is instead referring to roles that any participant in the distributed system may take on at different times: in this context, if a peer is acting as a server – servicing channel data get requests, for instance – its interactions with clients remain largely stateless.

client state into requests allowing channel *get* and *put* operations to be stateless and avoiding the need for a channel to keep track of its clients and the items they have fetched. The alternative, having time variables reference implicit client state, requires each channel to keep track of its clients individually and which items they have seen previously, severely complicating matters like moving between peers, replication and transparent interposition of caches.

At the programming model level, there is also significant value in providing abstractions high-level enough to capture the essence of a particular problem – since channels provide a single unified mechanism for data interactions with streams, the system has a unique vantage point from which it can monitor and adapt to overload conditions in an application-appropriate way. With lower-level non-stream-oriented messaging layers or various storage systems, such critical domain-specific information is spread across many system layers and components.

## *6.5 Evolution of the System*

The first full system implementation consists of three major distinct pieces: 1) the peer library implementation, 2) the supernode implementation, and 3) the front-end implementation. The peer library is the most complicated component by far, and it exists in two parallel implementations: one in native Java and one in native C (with some C++ convenience wrappers with default parameters and function overloading). The initial supernode implementation was piggybacked on the peer library, so all peers would also have supernode functionality available; this integration was later abandoned and the supernode functionality was pushed into a separate component. The front-end was first implemented as a separate C++ daemon. As the system progressed, the front-end was re-implemented in

Python to facilitate quicker development.[13]

**Erlang Supernode implementation:** Again, to facilitate quicker development and harness powerful existing functionality, the supernode implementation was moved to a separate component implemented in Erlang [199]. Erlang is a concurrent functional language well suited to network services and distributed programming. Since the supernodes cooperatively form a distributed directory of system-wide metadata, the goal of moving to Erlang was to harness the powerful Erlang-based Mnesia [138] distributed database. Mnesia works well for highly-available, replicated databases of moderate amounts of data, which fits the requirements of a system-wide metadata directory perfectly. The developers of CouchDB, an alternate Erlang-based non-relational document-oriented database, note that this is Mnesia's sweet spot: "[Mnesia] works best as a configuration type database, the type where the data isn't central to the function of the application, but is necessary for the normal operation of it" [7].

The second full implementation of the architecture features more extensive changes: the client library was rewritten in Java using a different communications architecture and the supernodes and front-end were merged into a single component, using an existing distributed coordination service called ZooKeeper [3].

**Supernode and front-end:** The original implementations of the supernode and front-end were built from scratch. In the years since their development, Google's suite of externally published core distributed systems technologies, including MapReduce [71], the Google Filesystem [89], Bigtable [64], Sawzall [163] and the Chubby lock service [59] have been

[13]The Python front-end has fewer than 200 lines of code, while the C++ front-end has less functionality and is more than 800 lines. The cost of experimentation and functional changes is significantly lower in a high-level language like Python, which is a boon for system evolution.

cloned by the open source community under the Apache Software Foundation's Hadoop [2] project umbrella.[14] This ongoing effort has produced a set of relatively robust pre-built components for supporting various distributed applications. In particular, ZooKeeper subsumes most of the functionality of the supernode and front-ends. The ZooKeeper project describes the component as "a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services" [3].

**Communications Architecture:** The second complete incarnation of the peer library uses Netty [13], a high-performance, event-driven, non-blocking networking framework for constructing network servers and clients. Leveraging this existing framework reduces the codebase by nearly 25% and improves maintainability and performance. As mentioned in Section 6.3.2, the runtime's persistent channel connection management strategy is also different in the new client library incarnation.

In the new architecture, channel operations normally go through a peer-wide connection pool. Persistent connections to other peers are created on demand – the pool grows when no free connections are available for pending operations and excess connections are culled over periods of inactivity. Such persistent connections are peer-oriented instead of being associated with specific channels (or specific threads); if peer A hosts channels 1 and 2, a persistent connection to peer A can be used for communication with either channel as required. Like the earlier channel-oriented cache design, the runtime's central management of connection pooling also provides a mechanism to use local channel replicas peer-wide (see

[14]Hadoop Core [2] is an implementation of MapReduce [71], HDFS (the Hadoop Distributed Filesystem) is Hadoop's Google Filesystem [89] equivalent and HBase is modeled after Bigtable. Pig [156] is inspired by Sawzall [163] and ZooKeeper [3] incorporates elements of the Chubby lock service [59].

Figure 13 in Section 4.3). If more flexibility is desired, special-purpose manual connection management is also still possible by explicitly creating and using `HostConnection` objects to perform channel-related operations.

The wire format was also changed to use Google's Protocol Buffers [15] data format and toolchain instead of the old XDR [80] format with the `rpcgen` toolchain (from SUN/ONC RPC [186]). See Section 6.6.2 for more on the wire format/protocol considerations.

## *6.6 Implementation Specifics*

In this section, we focus in detail on several specific implementation and concrete architectural issues of interest.

### 6.6.1 Channel Representation

The *channel* is the fundamental primitive in the programming model; consequently, it is important to consider the internal representation of channels to achieve adequate performance. Channels store a series of time-indexed items and two basic operations must be fast: 1) given a timestamp $t_{\text{lower}}$, find the first element with timestamp $t \geq t_{\text{lower}}$ and, 2) sequential access to adjacent elements. Balanced trees work well for the first requirement, and satisfying the second requirement can be achieved by using slightly more complicated variations of balanced trees where leaf nodes are linked for sequential traversal. Popular examples include threaded red-black trees and B-link trees [122], which also support relatively high-concurrency (but not lock-free) operations.

In fact, the first C-based implementation simply uses a simple (non-threaded) red-black tree plus a supplemental linked list: the tree is used to find the initial key position and the list is used for sequential traversal. Both data structures are protected using the same lock.

This solution was chosen for its simplicity, but it inhibits concurrent traversal.[15] An interesting data structure that also satisfies both properties (fast search and sequential access) is the skip list [166], a simple probabilistic data structure designed as an alternative to balanced trees. Owing to their relative simplicity, skip lists also lend themselves well to high-concurrency (and potentially lock free) use [165]. In fact, Java's JSR-166 [119] concurrent utilities enhancement adds a variety of high-concurrency data structures, and the concurrent ordered map implementation uses a skip list instead of a balanced tree. Doug Lea provides the following comment in the JSR-166 implementation of `ConcurrentSkipListMap`:

> Given the use of tree-like index nodes, you might wonder why this doesn't use some kind of search tree instead, which would support somewhat faster search operations. The reason is that there are no known efficient lock-free insertion and deletion algorithms for search trees.

Both our first and second-generation Java-based implementations use Java's concurrent skip list-based map as the central channel data structure. Figures 21 and 22 list a basic potential channel implementation in Java using a `ConcurrentNavigableMap` (part of `java.util.concurrent`) as the underlying data structure. This channel implementation is simplistic for illustrative purposes and does not have triggers, or support for persistence or blocking operations.

The example implementation is rather straightforward; the most complex operation is the `get` operation. The channel get operation first instantiates the lower and upper time bound parameters to concrete times (in case they are time variables) and then uses them to search the channel contents, returning items falling within the specified interval.

[15] Despite this fact, it still performs acceptably for our purposes. See Section 7.1 for microbencmarks.

```
public class LocalChannel {

  private final
     ConcurrentNavigableMap<Time32, Item> chanData;
  private final int gcInterval;

  // Constructor initializes chanData and gcInterval

  private void doGCCheck() {
     Map.Entry<Time32, Item> a = chanData.firstEntry();
     Map.Entry<Time32, Item> b = chanData.lastEntry();
     if(a != null && b != null &&
          ((b.getKey().getTvSec() -
               a.getKey().getTvSec()) > gcInterval))
       chanData.remove(a.getKey());
     }
  }

  public boolean put(Item i) {
    boolean ret = (null !=
                   chanData.put(i.getTimestamp(), i));
    doGCCheck();
    return ret;
  }

// continued in Part 2 (get method)
```

**Figure 21:** A Basic Java Channel Implementation (Part 1)

```
// continued from Part 1

  public ArrayList<Item> GET(Time32 lower, Time32 upper,
                             int limit) {
    ConcurrentNavigableMap<Time32, Item> tMap;
    ArrayList<Item> lst = null;
    Map.Entry<Time32, Item> a = chanData.firstEntry();
    Map.Entry<Time32, Item> b = chanData.lastEntry();

    if(a == null || b == null)
      return null;

    Time32 oldest = a.getKey();
    Time32 newest = b.getKey();
    Time32 l = lower.instantiate(oldest, newest);
    Time32 u = upper.instantiate(oldest, newest);

    if(null != (tMap = chanData.tailMap(l, true))) {
      a = tMap.firstEntry();
      lst = new ArrayList<Item>(16);
      while(true) {
        if(a == null || 0 < a.getKey().compareTo(u) ||
           (limit > 0 && lst.size() >= limit))
          break;
        lst.add(a.getValue());
        a = tMap.higherEntry(a.getKey());
      }
    }

    return lst;
  }

} // end class LocalChannel
```

**Figure 22:** A Basic Java Channel Implementation (Part 2)

**Reference Counting:** The C-based runtime implementation has the additional complexity of dealing with manual memory management in a data-intensive, concurrent application. As mentioned in Section 6.1, channels use reference counting to manage reclamation of data associated with items. The channel implementation must coordinate with the runtime's communication subsystem to make sure reference counts are maintained properly, and this entails the use of an additional data structure. A channel's internal *get* implementation simply returns an array of `struct iovecs`,[16] each of which holds a pointer to an individual item's data. To ensure this data is not reclaimed prematurely while it is still waiting to be sent over the wire, reference counts are incremented during the *get* call before returning the data to be sent.

An item may be removed from the channel data structure but it will not be reclaimed (`free`'d) until the reference count reaches zero. After the data is sent, the refcounts must be decremented so item data can be reclaimed when necessary. To facilitate decrementing reference counts after a *get* call, the channel implementation also uses a hash-table, mapping item data pointers back to item metadata (so that the reference count can be located and decremented). Figure 23 lists example code for the decrementing process to illustrate refcount management. The current implementation's hash table manages concurrent access with striped locks.

### 6.6.2 Wire Format & Protocol

Mirroring the improvements in mature, freely available distributed systems components noted in Section 6.5, several new IDL compilers (and corresponding binary data formats) have become available in the past year, including Facebook's Thrift [183], Cisco's Etch [1]

[16] Used in the scatter/gather POSIX IO calls, `writev` and `readv`.

```
/* iov is n+1 elements;
   iov[0] is header data, not item data. */
void chan_get_done(struct chan_descr *cd,
                   struct iovec *iov, size_t n) {
  size_t i;
  struct ht_node *htn;
  struct item *item;
  char *ptr_key;

  /* Decrement in-use counts on "live" items' data. */
  for(i = 1; i <= n; i++) {
    ptr_key = RAW_PTR_TO_DATA_PTR((char *) iov[i].iov_base);
    htn = ht_lookup_p(cd->data_tbl, ptr_key);
    if(htn != NULL) {
      item = htn->data;
      ATOMIC_DEC(item->in_use, &item->m);
    }
  }

  free(iov);
}
```

Note: `ATOMIC_DEC` is a macro that uses atomic operations or mutexes depending on compiler support. `gcc` version 4.1 and newer will use the `__sync_add_and_fetch` built-in.

**Figure 23:** Channel Post-Get Reference Count Maintenance

and Google's Protocol Buffers [15]. The first system prototypes of *temporal streams* use Sun's `rpcgen` [186] to automatically generate serialization code for common protocol metadata (control information, headers, etc.). Although it works better than hand-maintained serialization code, `rpcgen` is old and unsatisfactory in many ways. The IDL is clunky and the generated code is often awkward in modern languages. While maintaining D-Stampede [31] and later designing *temporal streams*, I investigated `rpcgen` replacements. In early 2006 I wrote, "we can provide tools for the automatic generation of serialization code from the CORBA IDL [153]. Toolkits/ORBs like The ACE Orb [101], Orbit2, and ILU [26] can be harnessed for their IDL compiler and CDR (Common Data Representation) codec functionality." Shortly thereafter I also looked at Flick [79] and several other solutions but found nothing suitable – ultimately, I was looking for a system with 1) a simple IDL, 2) lightweight generation of serialization code decoupled from RPC mechanisms and 3) cross-language support. Most of the potential solutions did not explicitly support generating pure serialization code that was not coupled with an RPC framework.

Ultimately, the unsatisfactory state of IDL compilers for generating serialization code to support generic distributed messaging interactions was recognized and subsequently addressed by various companies. Google's Protocol Buffers [15], released in July 2008, fits my three criteria exactly and the compiler also generates serialization/deserialization code that is relatively "natural" and easy to work with in its supported languages (C/C++, Java and Python). In addition, the system supports data versioning and adding fields to existing formats, essential features for long-running distributed systems. Compared to Sun/ONC RPC, the Protocol Buffers IDL is simple, modern and more easily maintainable; Figure 24 shows the difference between Protocol Buffers (upper half) and Sun RPC's IDL (lower half).

```
message ServerMsgReq {
  required ServerMsgCmd cmd = 1 [default = INVALID];
  required bool hasData = 2 [default = false];
  required uint64 host = 10;

  message CmdChanPut {
    required uint64 chnum = 1;
    required uint32 numit = 2;
    required Time32 timestamp = 3;
    required Time32 duration = 4;
    optional bool no_persist = 5 [default = false];
  }

...
```

---

```
struct chan_put_data {
   int size;
   int num;
   struct time32 timestamp;
   struct time32 duration;
   int no_persist;
};

union chan_cmd_data switch(short opcode) {
 case CHAN_GET:
   struct chan_get_data cgd;
 case CHAN_PUT:
   struct chan_put_data cpd;
...

struct chan_cmd {
  chan_cmd_data data;
  int hostid;
};
```

**Figure 24:** Protocol Buffers vs. Sun RPC IDL

**Protocol:** In addition to the serialization format (and related tools) for protocol metadata, the actual wire protocol has also evolved over time. The first prototype uses a straightforward format with known-length headers and length-prefixed data items – messages to/from the front-end and supernode are a fixed size (8K and 512 bytes, respectively). Channel-related messages, like *get* and *put* operations, have a fixed 64-byte header. For *put* requests and *get* responses, the header is potentially followed by a specified number of length-delimited data items. As mentioned in Section 6.1, each channel data item is prefixed with a 20-byte header containing three values: 1) a 32-bit length, 2) a 64-bit timestamp and 3) a 64-bit duration. As the code is parsing a request (or response), it reads the header and decodes it to determine the number of data items to expect. It then alternates between reading item headers and data; it reads an item header, decodes it and uses the size data to read the correct amount of item data. For illustrative purposes, a simplified code example is provided in Figure 25.[17]

While this method is straightforward, it requires at least two read system calls per data item. The next protocol revision allows the receiver to use scatter-input/vectored reads (`readv`) by providing multiple data lengths in the header and then batching items in chunks (as mentioned in 6.1). The channel operation header is fixed at 64-bytes and has enough room for ten 32-bit item sizes after other necessary metadata. Many requests will contain ten or fewer items, although that ultimately depends on the type of data an application will store in a channel. After the first ten, items are batched in increments of 32, preceded with a 128-byte array of 32 lengths. Figure 26 provides illustrative code for this protocol strategy.

The second full re-implementation of the system (described in Section 6.5) significantly

[17]This and other provided examples omit various corner cases and error handling for clarity.

```
...

tcp_read(sock, respbuf, sizeof(respbuf));
/* XDR decode respbuf into req. */

for(i = 0; i < req.items; i++) {
  tcp_read(sock, cmdbuf, CHAN_METADATA_SIZE);
  /* XDR decode cmdbuf into ihdr. */

  ALLOC_ARR(buff[i], ihdr.dsize);
  tcp_read(sock, buff[i], ihdr.dsize);
}

...
```

**Figure 25:** Protocol Decoder #1 Example

overhauled the networking code, using the Netty [13] communication framework and Protocol Buffers [15] for data serialization. Since Protocol Buffers uses variable length encoding and is not self-delimiting, the wire protocol changed from using fixed-length headers to using length-prefixed headers. Variable-length headers can significantly simplify migration to new protocol versions, because new fields can be added as needed; explicit lengths make upgrades of long running systems much easier. Since we are already using variable length headers, we include all data item lengths in a single data header payload. Like the last protocol, this structure allows scattering reads for item data. The new protocol format is in a comment excerpted from the actual code listed in Figure 27.

### 6.6.3 Runtime Interface Philosophy

A *temporal streams*-based application uses a client library providing the interface to the system runtime. One of the concrete system design goals is to provide unintrusive software

```
...

static int readn(struct iovec *iov, size_t niov,
                 size_t *ilens, void **buff) {
  size_t i;

  for(i = 0; i < niov; i++) {
    iov[i].iov_len = ilens[i];
    ALLOC_ARR(buff[i], iov[i].iov_len+CHAN_METADATA_SIZE);
    iov[i].iov_base = buff[i];
  }

  return tcp_readv(sock, iov, niov);
}

...

tcp_read(sock, respbuf, sizeof(respbuf));
/* XDR decode respbuf into req. */

items = req.items;
niov = MIN(sizeof(req.ilens)/sizeof(*req.ilens), items);
readn(iov, niov, req.ilens, buff);
items-=niov;

while(items > 0) {
  tcp_read(sock, cmdbuf, ITEM_SIZE_DATA);
  /* XDR decode cmdbuf into isz. */

  niov = MIN(sizeof(isz.ilens)/sizeof(*isz.ilens), items);
  /* req.items - items == number of items read. */
  readn(iov, niov, isz.ilens, buff+(req.items-items));
  items-=niov;
}

...
```

**Figure 26:** Protocol Decoder #2 Example

```
/*
 * Uses netty's ReplayingDecoder to decode the message
 * structure.
 * <hdr len> <hdr> [<data hdr len> <data hdr> <data>]
 *
 * The suffix in [] is optional depending on whether
 * this message type is supposed to contain data.
 */
```

**Figure 27:** Second Protocol Format Description (in code comment)

– in other words, the library should be entirely passive and easy to integrate into existing software, both at the level of an application's code and its build process. Some distributed programming system implementations (e.g., Stampede's [167] CLF messaging layer [161]) require extensive hooks into core application functionality, which can constrain application developers and make integration into existing software difficult. Other systems may require compile-time code generation or preprocessing (e.g., Flux [58]), and differentiated application functionality is essentially subordinate to the runtime, receiving callbacks at predefined points.

By my metric of intrusiveness, the runtime library should be passive and subordinate to the application functionality – the application should choose how and when to call the library entry points, rather than the other way around. Note that this does not mean that the library cannot keep its own background communications threads; it simply means that the application's own control flow is self-determined and unmolested. Code preprocessing or code generation techniques via little- or domain-specific languages are fine to use internally[18] in building the *temporal streams* runtime library as long as it does not "spill over"

[18] and, in fact, very convenient when developing communications systems

to the library user. In other words, the application developer should not have to run preprocessing tools on his own source code or substantially modify his software build process to use the *temporal streams* runtime.

**Language Support:** As mentioned earlier, the system has two separate native, but entirely interoperable implementations: one in C (with enhanced C++ bindings) and one in Java. While maintaining two different implementations in parallel and preventing incompatibilities is painful, previous experience with JNI [126] and various inter-language wrapper-generators has been similarly painful, except the pain is exacted on all users of the system rather than just the developers. Tools to generate multi-language bindings like SWIG (Simplified Wrapper and Interface Generator) [48] are convenient and critical in certain scenarios, but the generated interfaces are typically cumbersome. Mike Burrows notes that Google's developers also have similar experiences with Java/C interoperability [59]:

> Java encourages portability of entire applications at the expense of incremental adoption by making it some-what irksome to link against other languages. The usual Java mechanism for accessing non-native libraries is JNI [126], but it is regarded as slow and cumbersome. Our Java programmers so dislike JNI that to avoid its use they prefer to translate large libraries into Java, and commit to supporting them.

Ultimately, like Google's Java programmers, I preferred to provide a native Java implementation since it pays off for all users of the library. In addition, the fact that *temporal streams* programs are naturally multi-threaded and the runtime requires its own background threads complicates any use of JNI. While these problems are surmountable, the end result will be nearly as complex as a re-implementation and is likely to be much more fragile.

The effort required is better invested in a native Java implementation. As an added benefit, the native Java implementation is also significantly more advanced than the C implementation, due to the availability of mature and powerful high-level concurrency primitives from JSR-166 [119].

**API Issues:** One common issue in messaging middleware with a C language heritage is large, unwieldy APIs. Figure 28 contains an example of a core, commonly used Stampede [167] library call with ten parameters.[19] This phenomenon is not in itself atypical in general-purpose distributed programming systems because they attempt to provide highly general behavior; however, it often bloats user code with unnecessary complexity. The *temporal streams* libraries use Java and C++'s overloading facilities (and C++'s default parameters) to make the API as simple as possible when users are calling core functionality with very common parameters. Figure 29 shows some of the C++ convenience wrappers of *temporal streams* followed by the fully general, 11-parameter mega-function.

```
spd_error_t
spd_chan_get_item (spd_index_t            chan_index,
                   spd_time_stamp_t       desired_time_stamp,
                   void                  *client_buf,
                   spd_size_t             client_buf_size,
                   spd_bool_t             block_client_flag,
                   spd_bool_t             gc_flag,
                   spd_time_stamp_range  *time_stamp_range,
                   void                 **item_buf,
                   spd_size_t            *item_size,
                   spd_mod_name_t        *item_producer);
```

**Figure 28:** Stampede Channel Get Item Prototype

[19] This brings to mind one of Alan Perlis's famous programming epigrams: "If you have a procedure with ten parameters, you probably missed some." [162]

```
inline struct user_item
           conn_get_1(struct conn_endpt *chan,
                      struct time32 *lower,
                      struct time32 *upper,
                      int group = -1) {
...

inline int conn_get_1_i(struct conn_endpt *chan,
                        struct time32 *lower,
                        struct time32 *upper,
                        struct user_item *i,
                        int group = -1) {
...

inline int conn_get_n(struct conn_endpt *chan,
                      struct time32 *lower,
                      struct time32 *upper,
                      char **bufs,
                      size_t *bufsizes,
                      int max,
                      int *meta_out = NULL,
                      int group = -1) {
...

int chan_get_n_gen(struct conn_endpt *ce,
                   struct time32 *lower,
                   struct time32 *upper,
                   int *retval,
                   char **buf, size_t *bufsize,
                   int max,
                   int *metadata, int block,
                   int group, int get_type) {
...
```

`chan_get_n_get` is the original call wrapped by the previous convenience functions.

**Figure 29:** Convenience APIs

### 6.6.4 Complexity

Table 2 shows the complexity of different system components measured in physical SLOC (Source Lines of Code) as counted by the open-source SLOCCount tool [206]. The table is broken down into two columns: regular source and generated code. For example, the Erlang supernode implementation consists of 510 lines of hand-written Erlang code, and 1,006 lines of automatically-generated serialization code created by `erpcgen`, an ONC RPC IDL compiler that generates Erlang XDR serialization code from an IDL specification. Similarly, the C runtime library is broken down into general-purpose utilities (data structures, network utility functionality, etc.), IDL, the actual core *temporal streams* client library and the five storage backends plus common support files and IDL. Table 3 breaks down the storage components individually. In Table 3, `fs1`, `gpfs1` and `mysql` are our custom filesystem-based, GPFS [177] distributed filesystem-based and MySQL-based backends respectively. `pgsql` is a PostgreSQL-based database backend provided as an alternative to the MySQL-based backend, and `null` is a simple "no op" backend which throws away data it receives and returns no results for *get* operations. Table 4 shows the complexity of the second generation of the system detailed in Section 6.5.

**Analysis:** In Table 2, the C client library totals around 6k lines of C code excluding the storage backends. This includes all of the other levels of the persistence stack described in Section 6.2. By comparison, the first-generation Java client library is around 1,600 lines of code but does not implement the full persistent storage stack. Finishing this portion would probably add about 500 more lines of code. One original development goal was the use of significant auto-generation of communication management and protocol code. Although the serialization functions are generated by `rpcgen`, a significant portion of the

**Table 2:** Full System Complexity

| Component | SLOC | Generated SLOC |
|---|---|---|
| Supernode (Erlang) | 510 | 1006 |
| Front-end (Python) | 192 | - |
| Utilities (C) | 1544 | - |
| IDL (Sun RPC → C) | 231 | 1084 |
| C Client Library (C) | 4507 | - |
| 5 Storage Backends (C) | 2030 | - |
| Storage Backend Common (C) | 250 | - |
| Storage Backend IDL (Sun RPC → C) | 45 | 282 |
| C Runtime Totals | 8449 + 276 (IDL) | 1366 |
| Java Client Library (Java) | 1663 | - |

**Table 3:** Storage Backend Source Complexity

| Component | SLOC | Generated SLOC |
|---|---|---|
| `fs1` | 771 | - |
| `gpfs1` | 568 | - |
| `mysql` | 391 | - |
| `pgsql` | 271 | - |
| `null` | 29 | - |
| Common | 250 | - |
| Common IDL | 45 | 282 |
| Total | 2280 + 45 (IDL) | 282 |

**Table 4:** Second Generation System Complexity

| Component | SLOC | Generated SLOC |
|---|---|---|
| IDL (Protobuf → Java) | 115 | 5692 |
| Java Client Library | 1517 | - |

protocol-related code is still hand-maintained; this code is very regular in structure and could be generated from a more declarative description. Conservatively, re-implementing the C client library using custom code-generation tools or even the powerful C++ template metaprogramming features would likely reduce the code base by at least 1,000 lines. In addition, a significant portion of the "Utilities" code could be removed by using various STL data structures or some high-quality utility libraries such as Boost's.[20] Besides being more modern than C, Java's extensive standard library is another reason why the Java-based client runtime implementation is several times smaller.

Although *temporal streams* provides certain rich higher-level functionality – like time-based data stream data access, channel groups, persistence of historical data and pickling handlers – it is a relatively lightweight middleware substrate. As mentioned in Section 2.4, one of the goals of *temporal streams* is providing a simple primitive which is high-level enough to capture the essence of a problem but still low-level enough to be small, efficient and widely applicable. Part of achieving this goal is in providing a relatively lightweight runtime. Experiments in Chapter 7 and Chapter 8 quantitatively assess the performance overhead of our runtime.

By comparison, stream processing engines and database-oriented stream management systems tend to follow the opposite philosophy, attempting to provide a full stack – complete solutions spanning from low- to high-level. SLOCCount reports the Borealis [27] SDMS to be approximately 118k SLOC, primarily composed of 98k lines in C++ and 17.6k lines in Java. These statistics are generated from the Summer 2008 distribution

[20] `http://www.boost.org/`

of Borealis[21] with the `demo` and `test` subdirectories excluded and excluding the externally maintained Networking, Messaging, Servers, and Threading Library.[22] Similarly, the developers of IBM's Stream Processing Core reported the research code base size at 200k SLOC [34]. The stark contrast between the complexity of *temporal streams*'s code base and these systems' highlights the significant philosophical differences and goals. See Chapter 10 for more detail on related systems.

## *6.7 Architectural Enhancements and Design Alternatives*

In this section, we present a variety of alternate and extended features for an implementation of *temporal streams*. These enhancements vary from pure system-level improvements, such as HTTP-based networking interfaces (Section 6.7.1), to user-visible enhancements such as enhanced stream naming and metadata support (Section 6.7.2). Although some of the enhancements are quite extensive, none of them require changes to the abstract system structure as described in Chapter 5. These modifications are all relevant at the level of the concrete software architecture.

### 6.7.1 HTTP-based Networking

While the initial implementation of the *temporal streams* runtime focused on providing efficient networking and keeping the wire protocol compact, *temporal streams* can map rather naturally onto HTTP. Although an HTTP-based implementation would incur slightly more metadata overhead than custom wire protocols, there are many benefits to using a widely-used, standard protocol like HTTP. The biggest benefit is the wide availability of implementations – a huge variety of existing components can "speak" and understand HTTP. In

[21] `borealis_summer_2008.tar.gz` with an MD5 checksum of 552EE6F44B6EC97774C97818C13E8CFF.
[22] `http://nmstl.sourceforge.net/`

addition, by using a standard protocol properly, a system can automatically benefit from the universe of existing components like proxies, tunnels and inline caches as well as protocol decoders for analysis.

To map temporal streams on to HTTP 1.1 [84], consider implementing channel *get* and *put* operations:

- *get*($ts_l$, $ts_h$) on channel `chan1` – one can simply perform an HTTP GET operation with a URI encoding the timestamp lower bound, upper bound and channel (and any other ancillary options). Figure 30 shows an example channel *get* operation. The URI specifies a particular *temporal streams* application instance as a root domain component, followed by a path specifying a channel and timestamps. The timestamps may refer to concrete times or time variables.

  To facilitate proper caching behavior in the presence of time variables, the server can respond with a kind of temporary redirect (HTTP 302, 303 or 307); the redirected URI will specify the proper concrete timestamps. Responses to requests with concrete timestamps can be cached by clients or intermediaries.

- *put*(item, $ts_i$) on channel `chan1` – this operation can be implemented as an HTTP PUT or POST. Figure 31 shows an example channel *put* operation implemented as an HTTP PUT. Like *get*, the URI specifies an application and channel; it also specifies the concrete timestamp for the new item. If the user does not provide a timestamp, the system could use *now* at the time of the request and return a redirect HTTP response (e.g., 303) to indicate the true URI of the new item, timestamp included.

```
GET http://tsinstance/CHAN1&low=ts_l&high=ts_h HTTP/1.1

HTTP/1.1 200 OK
Cache-Control: private, max-age=0
Date: Wed, 29 Apr 2009 22:31:50 GMT
Transfer-Encoding: chunked
ETag: "5f8e38723ebbf6e01426879d360b3e77"
Content-Length: ...
```

- $ts_l$ and $ts_h$ will be some encoded representation of concrete times like `1241043627.015000` or time variables (e.g., `newest-0.015023`).
- `tsinstance` represents a particular application using temporal streams (i.e., a namespace encompassing all the channels associated with a particular distributed application).
- `chan1` represents a specific channel identifier within `tsinstance`.
- The server response disallows caching if requests contain un-instantiated time variables, but allows caching for requests involving concrete times.

**Figure 30:** Temporal stream get as HTTP GET

### 6.7.2 Enhanced Stream Naming & Metadata

The *temporal streams* abstract programming model defines stream data interactions and their semantics; it does not specify the overall distributed system structure or how streams are named, located and managed. It also does not constrain the content of data within streams – such functionality is left to the concrete implementation or application, which makes the programming model relatively flexible. In our architecture and implementation, we have described a relatively simple distributed stream directory allowing streams to be identified by names and system-generated unique identifiers. In addition, we have left streams as untyped or implicitly typed entities (any data type information is entirely

```
PUT http://tsinstance/CHAN1&ts=ts_i HTTP/1.1
Content-Length: 58438
Content-MD5: Q2hlY2sgSW50ZWdyaXR5IQ==
Content-Type: application/x-www-form-urlencoded
Connection: keep-alive
...

HTTP/1.1 200 OK
Date: Wed, 29 Apr 2009 22:35:32 GMT
```

- $ts_i$ is some concrete timestamp encoding.
- `tsinstance` represents a particular application using temporal streams (i.e., a namespace encompassing all the channels associated with a particular distributed application).
- `chan1` represents a specific channel identifier within `tsinstance`.

**Figure 31:** Temporal stream put as HTTP PUT

application-level). When channels use pickling handlers that switch between data formats, items must be tagged with internal type information to allow analysis code to determine the data format.

The potential design space involving enhanced stream location/naming/binding and stream-internal metadata (such as type information) is very large. In this section we will discuss a few alternatives design choices involving enhanced stream naming and metadata and their benefits.[23]

**More Expressive Stream "Subscription":** Our treatment of stream operations closely follows traditional distributed message queuing approaches: a client finds a particular

[23] Ultimately some of these alternate designs could be specified at the programming model level; currently, however, the programming model is flexible and leaves such details to the architecture or implementation, so they are presented in this chapter for continuity of presentation.

stream by unique identity and executes *get* and *put* operations on a particular unique stream. Publish/subscribe-style systems tend to support more expressive means of sending and receiving data; in such systems, users specify endpoints of interest in a more declarative way, typically via categories (topics) or constraints on specific message content. This paradigm is sometimes applied in distributed stream processing systems (see Chapter 10 for more context). For example, IBM's Stream Processing Core (SPC) [34] features a publish/subscribe-style stream naming system whereby consumers specify input data using a *flow specification*, which matches properties of streams including data types.

Such functionality is often most useful in higher-level stages of stream analysis aggregating data from many sources. For example, a video surveillance system may have many video streams; each video stream may have identical and independent first-level feature detectors (motion, background maintenance, foreground separation, blob tracking, etc.). Adding a new camera video feed to a running system is simple and doesn't affect other streams' first-level feature detectors. Higher-level analyses which aggregate results over multiple camera feeds, however, may need to be proactive about dynamically finding and fetching data from newly added streams or streams which suddenly become "interesting" based on dynamic information. This kind of paradigm can be supported in various ways in a realization of *temporal streams*.

- **Stream name wildcards** – One weak form of enhanced stream naming enabling easier dynamic subscription involves stream name "wildcards." For example, an application component could find streams matching the regular expression "`vid_chan.+`" where `.` implies "any character" and + signifies repetition of one or more times. Although this kind of matching is quite simple to implement, applications can use naming conventions to encode categorization data into stream names, giving functionality

similar to topic-based stream subscription. For example, an application can name all video camera channels with a prefix `cam_`, and prefix all background model output channels with `bg_`.[24] In the most basic realization of this functionality, however, the dynamic resolution of channels matching a particular wildcard is done at lookup and binding time rather than for each stream *get* or *put* operation. This could be combined with a strategy to periodically refresh the current set of streams by performing a new wildcard lookup.

- **Dynamic channels** – In a slightly stronger form of the previous strategy, the system can allow "dynamic" channels which are actually materialized views of separate component channels with tagged items indicating their original lineage. For instance, the dynamic channel named "`bg_.+`" might dynamically provide data from multiple background model channels. Performing a *get* on one of these channels would be equivalent to executing a sequence of get operations on each individual channel currently existing with names matching the regular expression. This strategy could also be used with channel categorization metadata independent of channel names – for example, each channel could be tagged with one or more application-meaningful categories.

Ultimately there is a rich space of design choices allowing ever more expressive selection and naming of streams, particularly in publish/subscribe systems and in various stream database efforts. While some applications certainly benefit from more dynamic stream "subscription" mechanisms, our programming model's view of streams lends itself to a certain style of constructing stream analysis applications which may not need such

[24] These channel names would also share common suffixes to indicate data lineage. For example, `bg_345` would be the background model derived from video captured camera channel `cam_345`.

advanced support. *Temporal streams* makes certain assumptions about the meaning of component streams. It is designed with a model of significant streams that typically originate from something with physical identity that must be processed to be interpreted – sensors' output, for example, or higher-level streams derived from non-trivial analysis on fundamental streams. On the other hand, pub/sub systems may view streams as more volatile predicates of interest over arbitrary application-level data. Both views lend themselves to different application construction styles; stream analysis applications can be constructed using both world-views, and particular application domains may be better suited to one or the other.

**Stream Type System:** Currently, channel item data is entirely opaque from the perspective of *temporal streams*. A realization of *temporal streams* could also provide a type system for channels whereby channel items would be associated with type information describing the format of item data. For example, one channel might contain video frames in 24-bit pixel-packed RGB format with a resolution of 640x480; another video channel might have grayscale JPEG compressed video frames with a higher resolution. This type information could be constant over a single channel, or vary on a per-item basis; current applications using temporal streams follow either strategy, depending on their specific requirements. Applications that rely on pickling handlers often tag each item with internal metadata describing the item's data format, because a persistent channel might have items in multiple formats – for example, uncompressed "live" data and compressed historical data.

The system could provide an official, built-in mechanism to associate user-specified type information with data items in channels since this functionality may be used with some frequency. Such functionality could inter-operate with system-provided hooks for

marshalling and unmarshalling code if items contain structured data. Some middleware solutions like Media Broker [144] use a hierarchical media type system in order to reason about and automatically reconcile data format mismatches by interposing content transformation functionality and changing sensor input parameters where possible. In fact, there is often implicit coupling between stream type data and stream "subscription" as discussed above. Many publish/subscribe systems specify data of interest based on attributes of structured data or type information. Again, there is a potentially large design space for type systems, some of which is explored in related work in stream databases and related systems.

**More Complex Registry:** Many of the alternative designs discussed above involve adding more complexity to the distributed metadata directory or "registry" component of the system.[25] Below we consider various other enhancements to these facilities involving other stream-associated metadata which could be stored in the registry.

- **Priority & Feedback** – In some applications, independent distributed analysis components may want to share priority information or other feedback about specific streams; the central metadata directory is a natural place to provide a shared "whiteboard" for storing arbitrary application-interpretable metadata associated with streams. Within the runtime system itself, storage backends (or pickling handlers) could use specific priority data provided by stream analysis components to better tune internal parameters.

- **Access Policy** – The metadata directory could also be used to store access or security policy information. Depending on the requirements, the actual interpretation of the

[25]This directory is stored cooperatively by the supernodes in the first iteration of the system and by ZooKeeper [3] in the second design.

policy could could be implemented entirely at the application level – for example, the application could allow any entity to retrieve channel data, but the data would be uninterpretable without a shared secret. Alternately, some access control mechanisms could be added and enforced within the runtime system itself. See Section 6.7.4 for more related to security and access control.

### 6.7.3 Enhanced Persistent Data Lifecycle Management

The *temporal streams* storage/persistence stack, described in Section 5.2 and Section 6.2, implements straightforward and useful stream persistence with pluggable storage backends and user-provided item transformation functions. In the current software architecture, the flow of persistent data is very linear – items move from live channels to persistent backends, and they are potentially transformed/modified during this transition. Free space management/storage reclamation and related ILM issues are delegated to particular storage backends. It could be beneficial, however, to "close the loop" and provide a set of hooks whereby storage backends could feed data back into the *temporal streams* stream persistence layers so items could flow through the system multiple times rather than once. This would enable several advanced features like allowing data re-transformation to further reduce the fidelity of streams as they age. It could also be used to provide multiple levels of directly cooperative hierarchical storage by allowing items to "age out" of one backend and re-enter the generic persistence layer to be sent to another backend. Currently, achieving this behavior requires advanced backends such a GPFS [177] which perform internal hierarchical storage management; closing the loop would allow completely independent backends to perform the same function, potentially using better higher-level stream information to guide system decisions.

The system architecture could also generalize the scope of live stream data "garbage" collection – currently, stream data is pushed to storage backends as it is garbage collected from the window of live stream data. This could be replaced with a more general "data lifecycle" management system which could uniformly handle decisions involving both live items or historical items on any of various storage backends. An orthogonal enhancement involves mediation between multiple persistent channels hosted at the same peer; currently each channel's data is transformed and flows to a storage backend independently. Decisions about shared resources are largely made independently rather than cooperatively, except where multiple channels feed data to common shared storage and the storage backend includes advanced mediation facilities (such as quotas, storage pools, IO bandwidth limiting, etc.). Coordinated persistence functionality could use the priority and feedback features mentioned in Section 6.7.2 to make better decisions about the relative priority of streams to an application and make more globally optimal resource decisions.

### 6.7.4 Security Policy

Security is an important but complex topic in distributed systems. As we have briefly mentioned in Section 6.7.2, there are a variety of alternate design choices for adding security and access control to our *temporal streams* implementation, and some of this functionality belongs in the metadata directory. Realistically, though, security and access control decisions affect the entire concrete system, and there is a large design space running the gamut from simple to elaborate. One of the reasons for the vast potential design space is the need to define threat (or attack) models – what kind of issues will the system actually attempt to protect against? For example, are independent analysis components within a single application mutually distrusting? Are there multiple trust domains within a single application?

Are we trying to guard against network data snooping or tampering by external entities? What about denial of service by external entities or denial of service within the system by erroneous or malicious participants? Going even further, is sensor input data trusted? Are analysis components' outputs trusted?

The first *temporal streams* implementation provides basic but weak security and access control. The metadata directory provides simple access control to prevent unauthorized entities from changing channel location data by generating a shared secret upon channel creation. The shared secret is used for authorization to modify channel parameters – the creator of a channel will receive this shared secret after a new channel is created. In addition, there is a pre-shared secret providing administrative power over the entire metadata directory. The ZooKeeper-based implementation uses ZooKeeper's [3] Unix filesystem permission-like Access Control List (ACL) mechanism.

One potential way to handle authentication is to rely on a protocol like Kerberos [148]. Using Kerberos, each channel can be treated like a network service and Kerberos client principals could represent individual peers or coarser-grained subsections of stream analysis applications. Point-to-point communication can be protected with traditional techniques such as TLS [76]. Secure transmission of data is complicated by issues such as the desire to multicast, latency requirements, performance overheads, dynamic changes in privileges and many other potential intricacies.

Ultimately, this whole area is a deep avenue of inquiry and requires both significant research and engineering. Some forms of access control can be added to the current system structure with no deep changes. Obviously "security" is a holistic issue and not just a matter of adding encryption and access control in key places.[26] At a high level, one

[26] Anderson and Needham explore the inherent subtleties associated with using cryptography in distributed

must determine the types of threats a system will defend against based on what threats are likely and what threats are feasible to defend against without unacceptably compromising system performance. One must also determine what data should remain confidential and where data integrity is important. In addition, the boundaries of trust and granularity of principals must be defined, based on both what is needed and what is feasible. These are all complicated questions and may depend on both the application scenario in isolation and the context of system deployment (i.e., a video surveillance application executed on an internally maintained private cluster versus a similar application executed on Amazon's EC2 cloud service). Since some aspects of security may be domain-specific and *temporal streams* aims to provide a lower-level substrate for many such live stream analysis applications, the system should be augmented to provide hooks to effectively support higher-level functionality while remaining a flexible building block. To suit its intended purpose and design philosophy, the *temporal streams* runtime should only provide the core security and access control mechanisms that are either widely applicable or cannot be properly implemented at a higher level.

### 6.7.5 Straightforward Enhancements

This section lists a few useful optimizations that would be straightforward to add to our current implementation, requiring only self-contained design decisions (i.e., no significant external impositions on the existing system structure).

**Temporal Prefetching:** The entire concept of *temporal streams* is based on the observation that time is critical semantic information in the interpretation of continuous streams.

systems [35]. They note that many security failures ultimately result not from a lack of cryptography but misapplication, such as reversing the order of signing and encryption or "designers [protecting] the wrong things." They also warn, "do not assume that [encryption's] use is synonymous with security."

Locality of reference is often used in computer systems to optimize performance – in particular, temporal and spatial locality. These assumptions are already used in the structure of channels – "live" (recent) data is kept readily in memory, while historical data is kept on secondary storage. We can potentially make similar assumptions about spatial locality when accessing stored data in channels.

In particular, we can try to perform storage prefetching based on time information – for example, if the channel interaction layer of the channel persistence stack processes a *get* on the interval $[t_1, t_2]$, we can actually retrieve and temporarily cache the interval $[t_1 - \Delta, t_2 + \Delta]$, guessing that an event of interest in the original historical interval $[t_1, t_2]$ may lead to further exploratory processing in nearby time intervals. Note that this is actually a form of spatial locality, but we call it *temporal* prefetching because the spatial "adjacency" relationships in a channel are directly mapped to temporal relationships in the context of *temporal streams*.

We can also perform prefetching of stored items based on gets referencing live items. If we detect *get* operations at the edge of the live/stored boundary, we can prefetch some stored items in anticipation of future interest. A huge number of potential enhancements to these prefetching techniques exist in literature on caches and memory subsystems, but some techniques, such as prediction of strided accesses, may rarely apply in most stream analysis applications. Ultimately some domain-expertise may be required to tune activation thresholds and parameters such a $\Delta$ in prefetching. An implementation of this functionality should allow applications to tune or disable the functionality as appropriate.

**Chained backends:** The simple and uniform interface to storage backends allows the straightforward construction of multi-level or chained backends. One can easily create a backend that accesses multiple levels of hierarchical storage in order; alternately, a composite backend layer can send requests to multiple storage backends simultaneously to

overlap loading time. To better support multi-level backends, we could augment the current `get_interval`, `put_item` storage interface with an additional call to query backend interval information. For instance, each storage backend could keep a quickly accessible cache of upper and lower bounds for stored channel data. This would allow higher levels of the storage stack to quickly determine which backends may contain relevant items and where new items should go. Chained backends would benefit directly from the improved persistent data lifecycle management functionality described in Section 6.7.3, allowing items to migrate between different levels of hierarchical storage as they age.

# CHAPTER VII

# SYSTEM-LEVEL METRICS

In this chapter we present a series of microbenchmarks and experimental results designed to highlight features of the *temporal streams* programming model and evaluate the performance of key primitives in our architecture. These experiments are targeted quantitative evaluations measuring specific system components in isolation – see Chapter 8 for full-system, application-based evaluations. This chapter is divided into two sections: 1) basic channel communication and in-memory indexing evaluations (Section 7.1) and 2) storage/persistence stack evaluations (Section 7.2). For uniformity, we use the C *temporal streams* runtime with the first binary wire protocol version for all experiments (see Chapter 6 for runtime details).

## *7.1 Channel Communication Architecture*

In this section we present a series of experiments testing the features of the distributed channel communication architecture and basic, in-memory channel indexing (see Section 7.2 for persistence architecture benchmarks). We present three sets of experiments, the first of which is a set of microbenchmarks designed to measure the local cost of channel primitives. Next we present a series of benchmarks to demonstrate that channels can scale well when many consumers are contending for access, measuring network performance. Finally, we demonstrate that channel groups perform as expected in a realistic application scenario.

**Channel Primitive Microbenchmarks:** Figure 7.1 depicts a set of microbenchmarks

assessing the cost of local channel operations. These microbenchmarks are run on a 2.2GHz dual-core "Denmark" Opteron workstation[1] with 2GiB of DDR-400 RAM running 64-bit Linux 2.6.17.1 with a preemptable kernel. We measure the cost of a local item retrieval (*get*) operation because it is slightly more complex than placing an item in a channel and requires a traversal of the same data structures (as well as relevant locking). Figure 7.1 shows the cost of retrieving one item by timestamp from a channel while increasing the number of items (by an order of magnitude each time – the x-axis scale is logarithmic). Each item is 64 bytes, although the size of the item does not affect the retrieval time for a local operation since item data is returned by reference. Each result is averaged over five runs and the cost of a single operation is derived from a measurement of 10,000,000 identical operations performed sequentially. Due to the internal data structures used for bookkeeping, the cost of retrieving an item from a channel will vary between different items; consequently, the cost is measured for three different timestamps (one near the beginning, middle and end in the temporal sense). Both graphs show the worst-case observed[2] and average values. When increasing the number of items in an interval, the get time increases approximately linearly with the number of items (Figure 7.1).

**Channel Scalability Benchmarks:** The remainder of the benchmarks are performed on a cluster of dual-processor 3.06 GHz Intel "Gallatin" Xeon nodes with hyperthreading enabled and 1 GiB of memory; each processor has 1MiB of L3 cache, 512KiB of L2 cache and the front-side bus runs at 533MHz. Each node runs 32-bit Linux 2.6.9 and the nodes

[1] Each core has an independent 1MiB L2 cache.

[2] Measuring the absolute theoretical worst case is complicated due to the simultaneous use of several different data structures with amortized access times. However, a brute-force test for smaller sizes (1001 and 10001) revealed that the absolute worst-case behavior did not deviate more than a few percent from those presented.

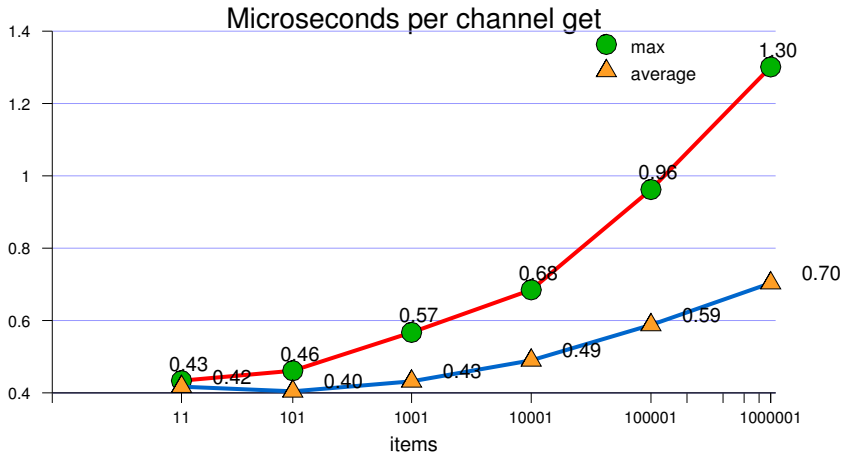



**Figure 32:** Channel get – Increasing Items

are interconnected with Gigabit Ethernet.

The channel scalability benchmarks test the ability of a single channel to serve many clients by scaling up the number of consumers and measuring contention. A single producer puts video frames into a channel at approximately 30 frames per second, and each consumer retrieves the *newest* available frame repeatedly, blocking while the last frame read is the newest frame available. As we scale the number of consumers contending for access to the same channel, we measure the number of dropped (i.e., skipped) frames by consumers. The number of dropped frames is a metric of scaling/contention that is directly meaningful to applications using the system.

Note that in these experiments, "dropped" frames are not dropped in the traditional sense – they are still available in the channel, but a consumer skipped them. The producer is putting a sequence of numbered video frames into a channel, and each consumer is reading the *newest* available frame in the channel as fast a possible. If contention is high, it is possible for the consumer to skip a frame. For example, a consumer may get frame #14 from the channel and the subsequent get may return frame #16 if the producer put both

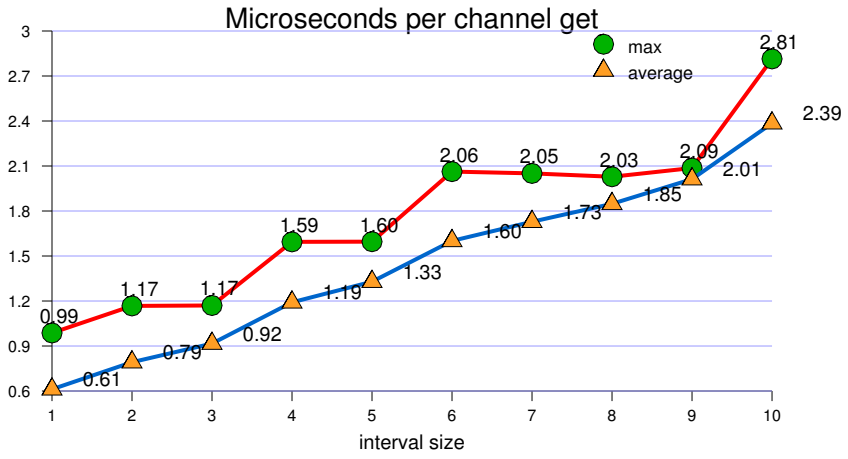



**Figure 33:** Channel get – Increasing Interval

frames #15 and #16 in the interim.

We scale the number of consumers and measure the number of skipped frames for two different kinds of video: low data-rate Motion JPEG (MJPEG) video and high data-rate uncompressed RGB video. Each dual-processor cluster node runs a maximum of four consumers and the single video producer runs on a separate, dedicated node. Each consumer negotiates a persistent, dedicated TCP connection to the peer hosting the channel (in this case, the producer). Each experiment configuration is run five times, and we show two key metrics: the sum of all frames dropped between all consumers (averaged over the five runs) and the maximum number of frame drops by any single consumer on *any* test run. We also provide a second graph showing the average number frames dropped *per* consumer (averaged over the five runs), but this is simply the sum of all frames dropped divided by the number of consumers. Motion JPEG is presented in Figure 34, and uncompressed RBG video is presented in Figure 35.

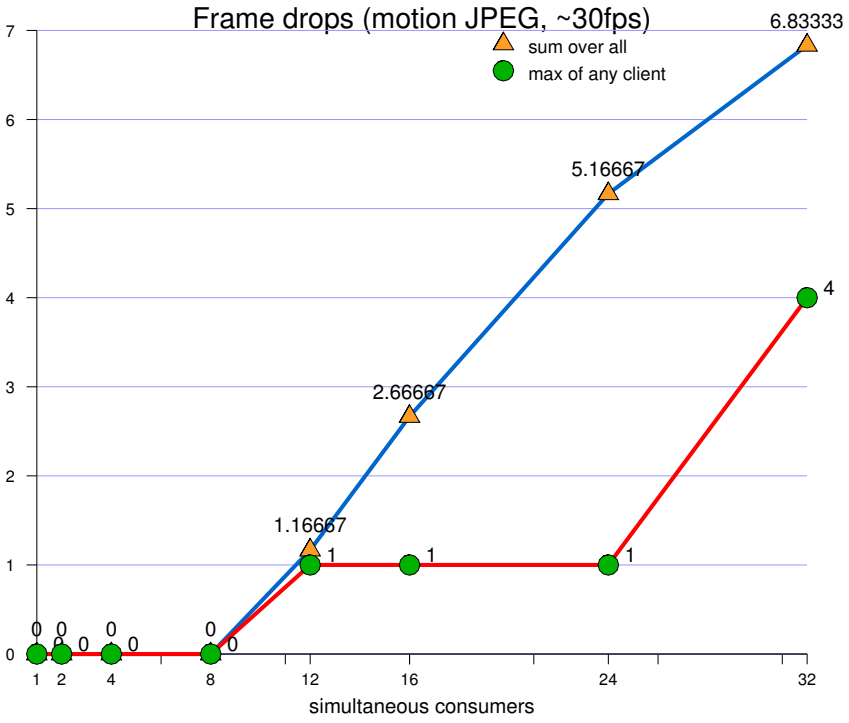
The total number of video frames in each experiment is 2312, thus each run lasts approximately one minute, 15 seconds. In our first configuration (Figure 34), the producer

puts frames of a Motion JPEG video stream into a channel at approximately 30 frames per second, and each frame averages about 18KiB. The data rate is relatively low, and the channel scales well. Even with 32 simultaneous consumers, the maximum number of frames dropped by any client on any run of the Motion JPEG test was four, and less than seven frames were dropped on average between all 32 consumers.

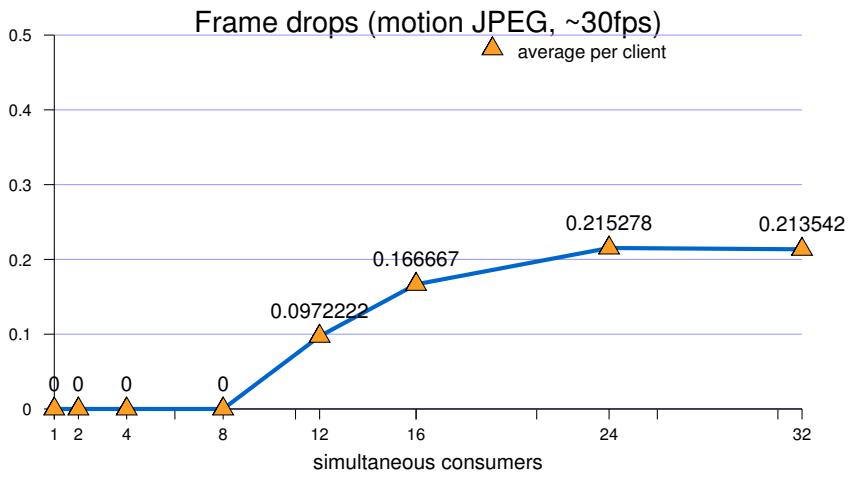
For the RGB configuration (Figure 35), the video frame size is a much larger 300KiB – a frame size appropriate for square uncompressed RGB video frames of 320x320. With this significantly higher data rate, the single un-replicated channel cannot serve as many consumers, but still scales quite well, dropping only 9.5 frames on average between all 12 consumers. With 16 consumers (not shown in the graph), the load is simply too high and the average consumer drops approximately 17% of frames, with the maximum loss rate for a single consumer at ~24% (dropping 549 out of 2312 frames).

The theoretical data rate for 16 consumers is greater than a single Gigabit Ethernet interface could support, but we can use channel replication split the load between two hosts. Figure 36 shows the results of using a single extra channel replica with the uncompressed video feed. The experiment setup is essentially the same as the previous experiment (Figures 34 & 35) with the addition of the channel replica running on a separate host: the first 11 clients are served by the original producer, with the remaining clients assigned to the replica. These results demonstrate that channels can scale well to serve multiple consumers, even with a relatively high data rate.

**Channel Group Benchmarks:** In order to assess the properties of channel groups, we utilize a kernel of a real application (extracted from *TV Watcher* [103]). This example uses two producer threads on the same peer: one produces MJPEG video frames at 30 frames per second, and the other produces decoded closed captioning text at the rate of two items

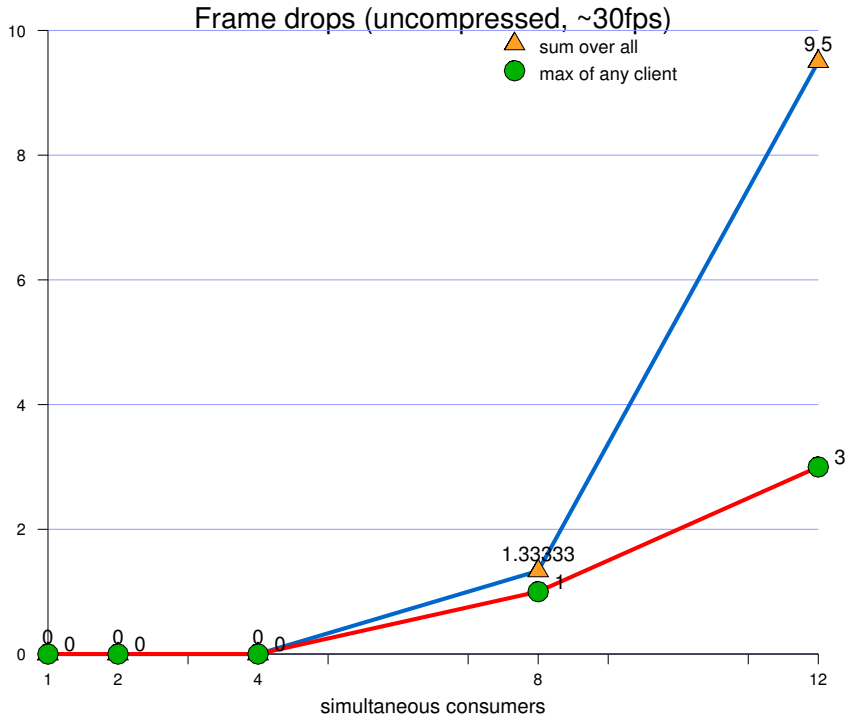



**(a)** Sum and max

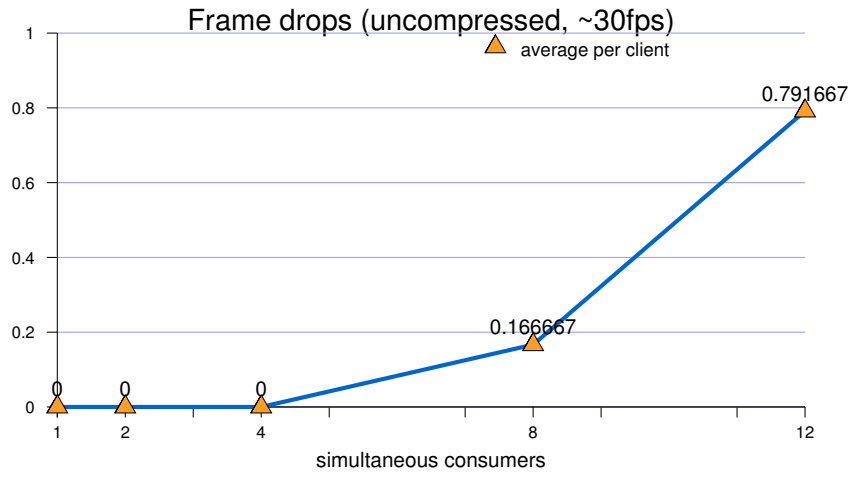



**(b)** Per consumer (sum ÷ consumers)

**Figure 34:** Channel Scaling Benchmarks – Increasing Consumers (MJPEG)

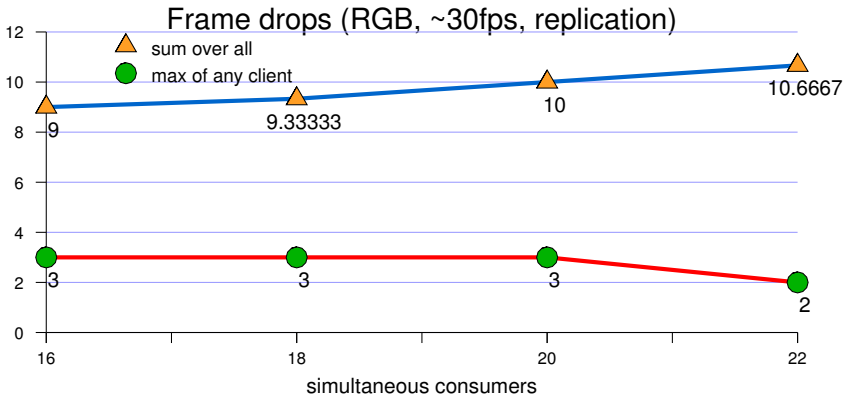



**(a)** Sum and max

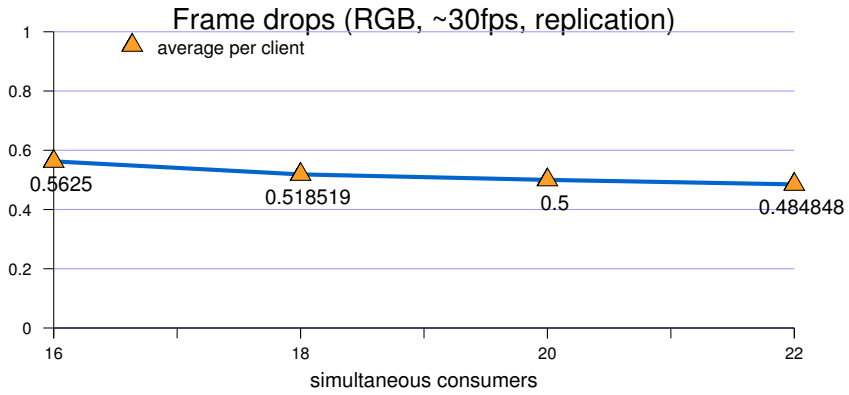



**(b)** Per consumer (sum ÷ consumers)

**Figure 35:** Channel Scaling Benchmarks – Increasing Consumers (uncompressed)




**(a)** Sum and max




**(b)** Per consumer (sum ÷ consumers)

**Figure 36:** Channel Scaling with Replication (1 replica)

per second (each item is 0.5KiB). A consumer on one machine retrieves video frames and a consumer on another machine retrieves closed captioning text items. In one case, the consumers both retrieve from channel descriptors that are part of a channel group synchronized dynamically with the reference stream *oldest*, while in the other case they simply retrieve new items as they are available. All three peers – the producer and two consumers – run on different machines and Figure 37 depicts the experiment setup. Figure 38 measures the average skew per frame in milliseconds. The total skew is calculated by taking the square root of the sum of the square of the differences between retrieval times of items from the two consumers.[3] Clearly channel groups lead to significantly lower skew and, although these results are somewhat obvious, the measurements simply provide a compact characterization of the performance of channel groups and show channel groups do provide viable synchronization in a realistic application scenario. While it is also true that the difference in skew between the two configurations can be made arbitrarily large by simply tweaking the relative rates of the two consumers, this set of parameters is taken from a real application. With channel groups, the visibility behavior of items in a channel operates like the diagram in Figure 4 (in Chapter 3). Since the programming model supports interval *get* operations, the video consumer with channel groups gets fifteen frames in a single operation when each new text item becomes available.

The previous experiments demonstrate that the key primitives in the programming model (channel operations) do not impose significant overhead, even with a large number of potential items in each channel. Additionally, contention does not significantly impede the scalability of channels to serve many consumers, even with high data rates. Finally,

[3]Each text item demarcates an "epoch" spanning approximately 15 video frames (until the next text item), and the differences are measured between video frames and text items of matching epochs.

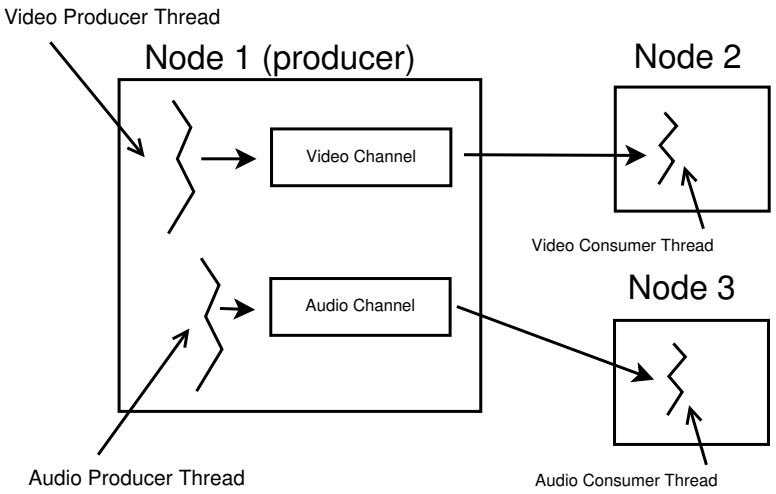



**Figure 37:** Channel Group Experiment Topology

channel groups operate as expected, reducing skew between get times of consumers on different hosts for items corresponding to the same temporal intervals, allowing consumers to operate on related streams in lockstep without explicit synchronization.

## *7.2 Storage/Persistence Architecture*

In this section, we perform several sets of targeted experiments designed to demonstrate the overheads incurred in the persistence architecture. We start with a relative comparison of the storage backends, the lowest layer of our stack. After that we use our most lightweight backend and target the higher layers, showing the overhead for *get* operations, performing storage scaling tests with pickling handlers and adaptive load shedding; we conclude with a benchmark showing the relative performance of live and stored *get* operations in both pathological situations with no locality and locality-friendly scenarios.

The first experiments are performed on an x86_64 Linux 2.6.22 host with two Intel Xeon E5310 processors (8 cores total, 1.60GHz) and 4GiB of DDR667 RAM. We added two dedicated 300GB 7200RPM Seagate 7200.8 SATA disk drives – one with a new XFS

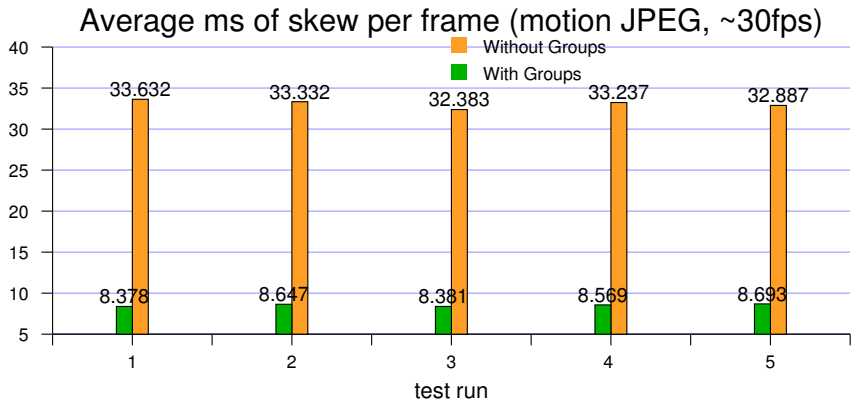



**Figure 38:** Channel Group Benchmarks

filesystem and the other with a GPFS filesystem (both spanning the entire disk). We use MySQL version 5.0.51a and GPFS 3.2.0. Our system binaries and related libraries are compiled with gcc-4.2.3 and both -O2 and -g flags.

**Single Producer Storage Backend Overhead:** The first set of benchmarks compares the relative overhead of the backends. We profile our storage backends with OProfile [123], a low-overhead, sampling-based system-wide profiling tool integrated with the Linux kernel; OProfile operates with un-instrumented binaries. Since OProfile provides whole-system profiling, we can account for execution time spent in external processes, like the `mysqld` daemon process (when using the MySQL storage backend). We run a single data producer process putting 18,000 300KiB RGB video frames at 30 frames per second into a persistent channel. During this time, all frames are stored using either the MySQL backend, the local filesystem backend (`fs1`), the GPFS-based distributed filesystem backend (`gpfs1`) or a null backend, which is a no-op (throwing the data away). For MySQL, the database server process is running on the same machine and stores its data on the same filesystem. Communication between the MySQL client library and the database server

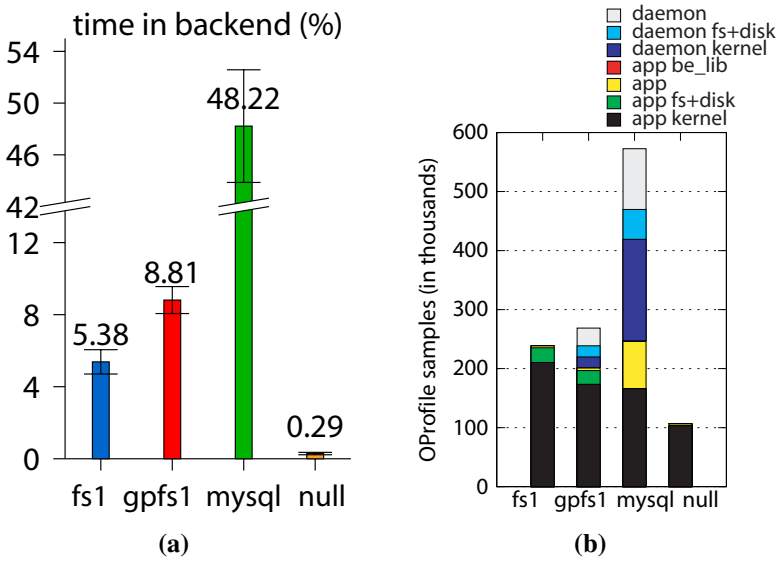



**Figure 39:** OProfile (single producer)

occurs via a domain socket. We average the results of five independent runs.

The OProfile results in Figure 39(a) show that the execution time overhead of the MySQL backend is very high relative to `fs1` (and `gpfs1`). Using another profiling tool, Callgrind (part of the Valgrind dynamic binary instrumentation tool-suite [147]), we notice that using the MySQL backend causes the producer to execute an order of magnitude greater number of instructions. See Table 5 for the Callgrind results of both the total instructions executed when using each backend and the relative percentage of instructions spent in the backend during item storage.[4] Callgrind attributes almost all of this overhead to `memcpy` called within `libmysqlclient_r`, the MySQL client library.

Examining the source, we find that the library explicitly copies all data inputs while

[4]One must take the Callgrind results with some care, however: Valgrind dynamically interprets a binary on a virtual processor, and it does not have the ability to give accurate cycle or time estimates. Obviously executing an order of magnitude more instructions will lead to significant extra overhead, but the percentage of executed instructions does not directly map to an equal proportion of execution time; system call time also cannot be accounted for by Callgrind.

marshalling them; this extra data copying significantly increases the overhead of the MySQL backend, making it unsuitable for high data-rate streams. This simply highlights the fact that the MySQL client libraries were not designed for this kind of workload as it is atypical for traditional relational databases. Additionally, our OProfile results indicate that even though the MySQL library cost is quite high, the relative cost of work done on behalf of the backend in the external `mysqld` process is even higher: if we consider both the time spent in `mysqld` and the actual data producer binary, we find that `mysqld` accounts for 82% of the time (with $\sigma = 0.36\%$). The overhead of the `null` backend is simply measuring the cost of executing a function call in a dynamically linked library. The relatively higher variance of the OProfile measurements can be attributed to its sampling-based approach, versus Callgrind's dynamic binary interpretation approach.

To get a more complete picture, we consider both user and kernel time accounted to the applications in question; since `fs1` and `gpfs1` are thin layers over the filesystem, much of the work will be done in the filesystem, accounted to system time. Figure 39(b) shows cumulative kernel and application results, broken down into kernel time, filesystem and disk kernel time, application time and application backend library time (when applicable). These results are presented for both the data producer process using *temporal streams* (`app`) and the external daemon process when present (`mysqld` for MySQL or `mmfsd` for GPFS). Results are in thousands of samples, each sample representing about 100,000 core CPU cycles worth of CPU time. Although the division between filesystem/disk kernel time and other kernel time is going to be somewhat imprecise, these results are simply designed to show relative trends between our storage backends. Again, the total overhead of the MySQL backend is very high and this is expected since data is copied within the address space of the data producer, sent to the `mysqld` process via IPC and then committed to disk.

**Table 5:** Callgrind (single producer)

| Backend | `fs1` | `gpfs1` | `mysql` | `null` |
|---|---|---|---|---|
| Total Instructions (Millions) | 128 | 143 | 5,477 | 126 |
| Instructions in Backend | 1.99% | 1.83% [5] | 97.50% | 0.03% |

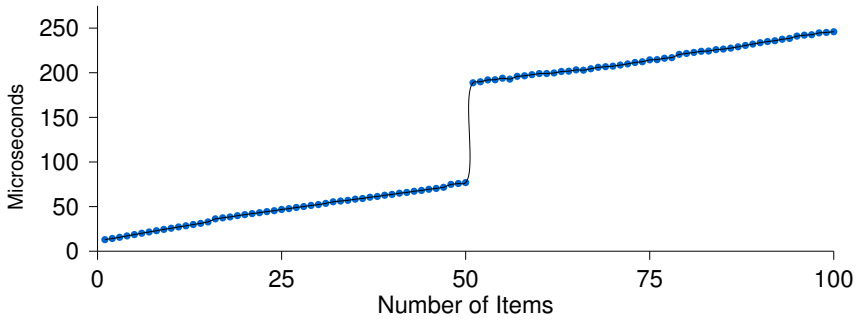
NOTE: *Instructions in Backend* percentages do not include initialization instructions.

Ultimately this isn't a fair comparison because MySQL is not designed to be a storage backend for bulk streaming data and has a lot of other rich functionality; additionally, neither tool accounts for idle time waiting for I/O. However, these results validate our decision (and intuition) to build lighter-weight storage backends – `fs1` is only about 600 lines of C code and `gpfs1` is similar. Although we haven't presented the read cost of the MySQL backend, it is clear that it will also incur client library and IPC overheads.

The remaining experiments in this section are performed on an x86_64 Linux 2.6.24 host with an Intel Core 2 Duo E6750 (2.66GHz) processor and 4GiB of DDR667 RAM. We use the `fs1` backend (on a dedicated 300GB Seagate 7200.8 drive with XFS) since it has the lowest overhead and no external dependencies.

**Single Consumer Get Overhead:** This experiment demonstrates baseline retrieval overheads of the storage layer with `fs1`. In Figure 40, we measure the cost of a *get* interval operation as we increase the maximum number of items in the interval to include stored items. We place 100 items in an eagerly persistent channel (using the `fs1` backend) which will hold up to 50 live items. Each item is 1024 bytes and retrieval is performed over loopback TCP/IP networking. Each *get* is performed 10,000 times and we report the per-get

[5] Although it may seem anomalous that the GPFS backend executes more total instructions but has a lower percentage of instructions spent in the backend, this is because the total counts includes initialization, while the percentage does not. The GPFS backend logic is slightly simpler when storing items, but sets sentinel entries in a large multi-level index up-front during initialization. The one-time initialization cost of the multi-level indices lead to a higher total instruction count.



50 live items, 100 stored items (half overlap)

**Figure 40:** Cost of get operations with an increasing number of items

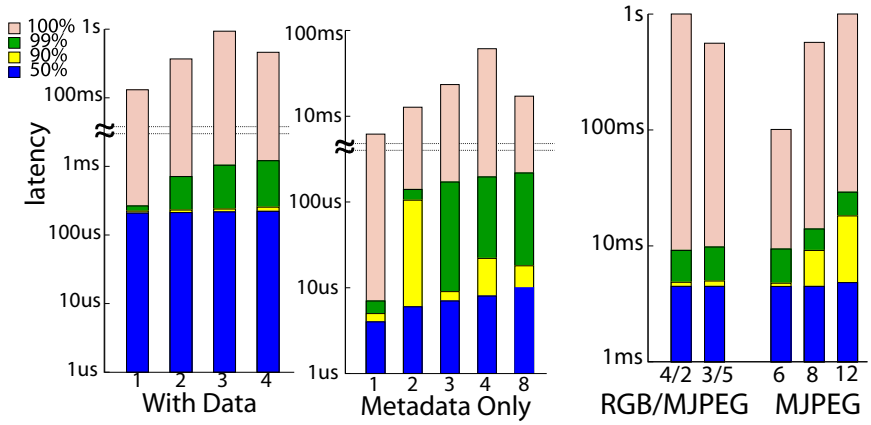
averages (i.e., measured time / 10,000); the values are averaged over five runs (the standard deviations are less than 1% and thus not drawn on the graph). In the figure, *get* operations scale roughly linearly with the number of items requested until items must be fetched from the storage backend. At that point, each operation incurs a fixed cost of approximately 118 microseconds, and the roughly linear trend continues – obviously the additional cost of accessing stored items will vary widely depending on the storage backend and underlying storage media, but these figures show baseline overheads for `fs1` (when all data is in buffer cache). This shows the basic cost of the extra logic for persistent channel data, including backend code.

**Multiple Stream Scaling:** This experiment shows how the `fs1` backend and our persistence architecture scale with increasing I/O rates by scaling the number of concurrent streams committed to the same disk. Figure 41(a) shows the results of multiple persistent channels simultaneously saving data to the same local XFS partition using the `fs1` backend with a chunk-size of 144MiB. Each channel is filled by a producer putting 300KiB RGB video frames at 30 frames per second, and the experiment runs for 36,000 items in each channel (20 minutes at 9MiB/sec per stream). We scale the number of concurrent producers

and show results for the normal configuration as well as results where data writes simply go to a file descriptor which throws away the data (`/dev/null`) – since the local disk will bottleneck long before other components, "no op" disk writes let us isolate the overhead of other pieces of our architecture. We modified the backend to get the current time after an item's data is written out and modify the item's stored timestamp to provide an estimate of the total latency from the time it arrives in the channel to the time it is written out. We also set the level of queuing in the generic persistence layer to one, so each item is sent to the backend as soon as it arrives to the generic persistence layer.[6] We present the results of item latencies in the form of several statistical percentiles (50%, 90%, 99%, 100%) because the general distribution is hard to characterize with a single number. For each percentile, we present the maximum among all producers. The vast majority of items have small latencies and then median times are quite low, but heavy I/O tends to induce a small tail of extreme outliers, particularly when the data rate streaming to disk is high (note the graphs' log scale and broken axes). The 99th percentile latencies seem to be primarily influenced by the amount of filesystem traffic and contention between multiple producers writing to a common disk. The absolute worst case measures (100%) have a high variance and are less meaningful across tests, because they are determined by a single high reading (i.e., the highest latency item during the entire run among all producers).

**Multiple Stream Scaling with Pickling Handlers:** The next experiment shows how applying pickling handlers to producers effectively reduces the data rate of streams committed to disk, enabling us to scale up the number of streams. We cannot reliably commit five

[6]This somewhat overestimates the overhead of the generic persistence layer because some of the inter-thread locking and signaling overhead is amortized over multiple items waiting in a queue; however, completely removing the queueing mechanism would underestimate it, so we left in the superfluous queuing.




**(a)** RGB video streams **(b)** with JPEG pickling

NOTE: These graphs Y-axes are plotted on a log scale, and Figure 41(a) also features broken axes. See Tables 6 and 7 for the raw data.

**Figure 41:** Item latencies by statistical percentile

**Table 6:** Raw Data: RGB streams – Item latencies by percentile

| N | With Data (by percentile) | | | | Only Metadata (by percentile) | | | |
|---|---|---|---|---|---|---|---|---|
| | 50% | 90% | 99% | 100% | 50% | 90% | 99% | 100% |
| 1 | 208$\mu$s | 223$\mu$s | 265$\mu$s | 0.131s | 4$\mu$s | 5$\mu$s | 7$\mu$s | 6.19ms |
| 2 | 212$\mu$s | 231$\mu$s | 711$\mu$s | 0.368s | 6$\mu$s | 105$\mu$s | 140$\mu$s | 12.73ms |
| 3 | 217$\mu$s | 239$\mu$s | 1.04ms | 0.934s | 7$\mu$s | 9$\mu$s | 171$\mu$s | 23.39ms |
| 4 | 221$\mu$s | 254$\mu$s | 1.21ms | 0.461s | 8$\mu$s | 22$\mu$s | 196$\mu$s | 61.29ms |
| 8 | - | | | | 10 $\mu$s | 18$\mu$s | 218$\mu$s | 17.12ms |

concurrent 9MiB/s streams using `fs1` with our particular hard disk and XFS, so we configure a pickling handler to compress each 300KiB RGB video frame into a JPEG image. The average JPEG size is 20K, a fifteen-fold reduction in data committed to disk. Figure 41(b) shows the results for runs with 6, 8 or 12 producers all doing JPEG compression, and a mix of RGB and JPEG producers. The item latency now includes a JPEG compression step, performed by `libjpeg6b`, so the median item latencies are ~4.5ms versus ~210$\mu$s without the added compression and creation of temporary items. The raw measured cost of the JPEG compression by itself (without dynamic allocation of items or buffers) is ~3.7ms per frame on average. Although the data rate of 12 MJPEG streams is still less than a single RGB stream, each producer requires at least 270MiB of memory to hold 30 seconds of RGB data in the live channel (plus some extra memory for temporary JPEG items), and we run into some physical memory pressure around 14-15 streams. We could reduce the number of seconds of live data that each channel holds to add more producers, but we eventually hit a CPU bottleneck for JPEG compression before the disk bottlenecks. If we look at the all JPEG producer runs versus the mixed runs, we see that the 99th percentile latencies are now more indicative of CPU contention versus disk contention; since we present the maximum value over all producers for each percentile and compression adds significant latency in the critical path for all JPEG streams, the storage latency for uncompressed items will generally be overshadowed by JPEG items. Again, the 100th percentile measures are less meaningful.

**Dynamic Load Adjustment with Pickling Handlers:** This experiment shows how the architecture can dynamically adjust to overload conditions. By measuring operation latencies in the generic persistence layer, the system can react by adding pickling handlers if the disk is overloaded or removing/changing them if the CPU is overloaded. The user could

**Table 7:** Raw Data: RGB & MJPEG streams – Item latencies

| N | RGB | JPEG | With Data (Percentile) | | | |
|---|---|---|---|---|---|---|
| | | | 50% | 90% | 99% | 100% |
| 6 | 4 | 2 | 4.47ms | 4.85ms | 9.14ms | 1.04s |
| 8 | 3 | 5 | 4.47ms | 4.97ms | 9.78ms | 0.56s |
| 6 | 0 | 6 | 4.46ms | 4.76ms | 9.40ms | 101ms |
| 8 | 0 | 8 | 4.47ms | 9.13ms | 13.98ms | 0.57s |
| 12 | 0 | 12 | 4.82ms | 18.13ms | 29.05ms | 1.07s |

also provide several pickling handlers to compromise between stored item size and computational cost. In our current prototype we've implemented a simple proof-of-concept to illustrate the possibility of dynamic load adjustment: currently we only consider disk load and a single pickling handler, but if the item latency starts to increase heavily, some number of consumers automatically switch to using their pickling handlers until the overload is resolved. We ran successful tests starting with 6, 8 and 12 RGB video producers with JPEG pickling handlers; in all initial configurations (6, 8 and 12 uncompressed video streams), the load is too great for the local disk and the system would normally fall behind and never recover without removing producers. Figure 42 shows the item latencies for a single producer of the 8 producer run before and after it switches to JPEG frames. The system is initially overloaded and latencies spike to multiple seconds since the data rate is too high for the disk; the system detects this overload and enables enough pickling handlers to recover, returning the latencies to around 5ms.

**Mixed Stored/Live Reader Workload:** In order to demonstrate the performance impact of accessing stored versus live items, we vary the percentage of *get* operations requesting live versus stored items and measure the time to perform 10,000 *get* operations. Again we use 300KiB RGB frames and perform *get* operations which request 50 items from a point

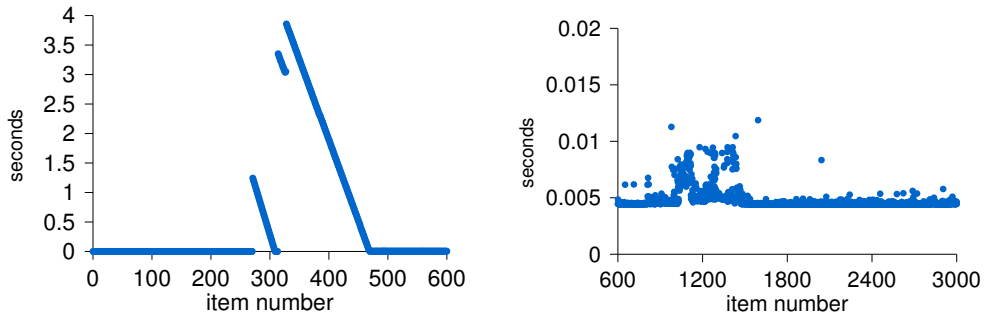



**Figure 42:** 8 Producers – Latency Before/After Adjustment

in the channel determined by a probability distribution. 72,000 items are placed into the channel with a storage backend of `fs1`, and the last 200 items will stay in the live channel. Since the size of all of the items is ~20.6GiB, the full set of channel data is much larger than can fit in memory. We measure the cost of gets of exactly 50 items from some random point in the channel (containing all stored or all live items), and we limit the transferred data of each item to 100 bytes to eliminate the network transfer overhead and emphasize the overhead of stored data retrieval (all data is still read from disk when stored items are fetched). We vary the percentage of requests for live items from 0 to 100 and measure the total time to complete 10,000 requests with three different distributions – a uniform random distribution, a Zipf distribution ($s = 2.0$) and a binomial distribution ($p = 0.5$). The uniform random distribution exhibits no locality and rapidly bottlenecks by the raw speed of the disk. Both the Zipf and binomial distributions exhibit a lot of locality and thus benefit from caching, scaling much better (in fact, their differences are too small to see on the graph scale). Figure 43 shows the average per-get time for the distributions (each point is also averaged over five runs). Although none of these test configurations are realistic models of an actual application, which might have many different clusters of "popular" historical data based on detected events, it does show the gamut of scaling behavior between

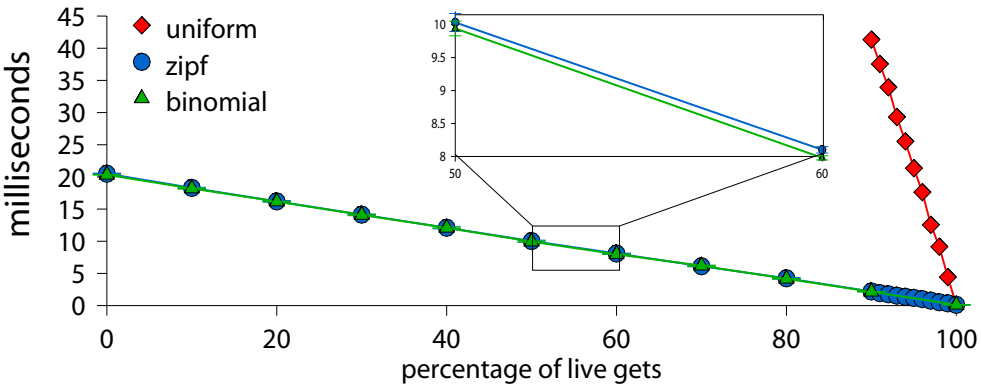



**Figure 43:** Per-get time with historical query distribution

pathologically bad and more locality-friendly workloads. Real workloads should fall in-between these extremes.

**Other Measurements:** Although the results are not presented here graphically, we have measured the effect of varying the `fs1` chunk size parameter. The results are largely unsurprising: a smaller chunk size increases the number of separate files created for a single stream. With high data-rate streams, these additional and frequent filesystem metadata operations are very expensive. Increasing chunk size improves performance to a certain point and then flattens out.

These system experiments show that the persistence architecture and primitives can be implemented in a lightweight manner, scaling to store relatively high data rate streams.

# CHAPTER VIII

# APPLICATION-BASED EVALUATION

In this chapter, we will use motivating applications to evaluate *temporal streams* in the context of realistic application scenarios. We perform both quantitative and qualitative analysis: we describe a series of performance experiments and discuss the particular application characteristics supported by *temporal streams*. Our three primary examples are designed to highlight key properties of the programming model and system. Besides our research group's direct experience, all of our motivating applications are related in requiring significant signal processing on heavyweight, unstructured but data-rich streams like video. Furthermore, they illustrate different aspects of *temporal streams* spanning from low-level processing to higher-level inferencing to broad system architectural properties.

In particular, the airport surveillance scenario highlights the use of historical data through persistent streams. The traffic monitoring scenario highlights the use of time-based synchronization: a channel group is used to synchronize the output of background maintenance with the raw video stream from which the background is produced, providing input for traffic density estimation. Finally, the port asset tracking scenario deals with the relationship between higher-level inferencing and time-oriented data access. After our three application evaluations, we briefly introduce a fourth scenario currently in development to show where *temporal streams*'s support of loosely coupled distributed systems is beneficial in practice.

**Roadmap:**

- Section 8.1 – We describe the three application scenarios at a high level, including relevant background information.
- Section 8.2 – We describe the implementation of each application scenario on *temporal streams*. We describe the *temporal streams*-based components of each scenario in detail as well as salient implementation details, including code samples. We also compare the scenarios.
- Section 8.3 – We perform quantitative performance analysis of the airport and traffic application scenarios and analyze the results. The application components are abstractly described in Section 8.2 independent of any deployment; in this section, we will describe the experiment-specific configurations of these components, including the connection topology, the assignment of components to physical nodes and the data inputs and parameters used.
- Section 8.4 – We evaluate the overall picture of the applications in the context of *temporal streams* given our results. Here we also provide our qualitative discussion of our three scenarios.
- Section 8.5 – We introduce a nascent project using *temporal streams* as a fourth concrete application scenario.

## *8.1 Applications*

In this section, we will describe three application scenarios at a high level, giving relevant background information. In all three cases, the scenarios are based on real-world problems;

we have toured all of the relevant sites and gathered information on current processes and technology. Our application examples are based on conceptualized next-generation systems with higher automation or more advanced analysis than current existing systems, but the computation is applied to existing, required processes already taking place manually.

### 8.1.1 Airport Surveillance

Airport surveillance is a timely but controversial topic; nonetheless, using such surveillance as a motivating example is practical because the scenario is familiar and easily accessible. Atlanta's Hartsfield-Jackson Atlanta International Airport[1] is consistently one of the world's busiest airports. As of mid-2007, the airport has around five hundred PTZ closed-circuit cameras with analog feeds to the main security control room. The control room has three to five employees monitoring a small subset of feeds selected manually, although all feeds are recorded for potential retrospective analysis. Their system is not automated in any substantial way, so our application will model a conceptualized next-generation automated system providing basic stream-of-interest detection to enhance employees' monitoring effectiveness.

The system will use common, computer vision-based higher-level feature detectors to pick streams of interest. For example, a face detector will extract faces from camera views; the face detector is actually a trained instance of a more general and widely-used object-detection technique [198]. In addition, optical-flow based motion detection will isolate areas of high interest. The raw streams' historical data are also stored in a compressed form, and simulated historical and live queries are performed to model human interest.

[1] `http://www.atlanta-airport.com/`

### 8.1.2 Traffic Monitoring

Traffic monitoring and airport surveillance share similar lower-level streams and sensors but differ in higher-level analysis and goals. In both cases, a moderate number of fixed video cameras[2] are positioned throughout an environment (e.g., an airport or metropolitan roadway system). The difference is mainly in the higher-level feature detection and analysis computation performed – the higher-level goals of monitoring and patterns of analysis are obviously domain-specific.

The Georgia state Department of Transportation[3] runs a central Transportation Management Center (TMC) with several satellite Transportation Control Centers (TCC) to monitor and respond to roadway issues. Their sensor system consists of over 1400 fixed black and white cameras for traffic density estimation (VDS – Video Detection System cameras), plus over 400 color PTZ CCTV cameras for monitoring. Queries come in from operators at the transportation centers, the Georgia Navigator website, and the `*DOT` traffic information line.[4] At the various transportation centers, operators take calls and monitor traffic cameras to detect incidents and direct emergency response.

Currently, the roadway video feeds are only used for viewing by human personnel and not for automated analysis; the specialized, fixed VDS cameras provide traffic density information. Due to the large volume of data and the nature of traffic monitoring, video feeds are not archived. In the TMC, a large display shows three camera feeds associated with the most important incidents as determined based on severity and recency. Our application

[2]Fixed in the sense of affixed to some sort of structure. Fixed cameras may still have operator-controlled pan, tilt and zoom (and, in fact, do in both of our scenarios).

[3]`http://www.dot.state.ga.us`

[4]See `http://www.georgia-navigator.com/about` for an overview of Georgia's Intelligent Transportation System (ITS).

will model a conceptualized future system that uses vision-based analytics on the camera video streams to estimate traffic density and pick streams of interest.

The computation includes a background maintenance algorithm which generates a background model to assist in detecting moving roadway objects and also in detecting breakdowns and other obstructions. Vision-based foreground separation is used for traffic volume estimates. A modified scene cut detection algorithm is used in coordination with the background maintenance to determine unanticipated camera angle changes or camera obstructions.

### 8.1.3 Port Asset Tracking

Operated by the Georgia Port Authority, the Port of Savannah[5] is one of the fastest growing and busiest ports in the nation. In 2008, the Port handled 8,000 truck transactions and 20,000-25,000 containers per day. Efficient logistics and coordination are critical with such high levels of cargo traffic. Tracking cargo and monitoring daily operational flow is important because anomalies can lead to costly delays and lowered efficiency. While the Port is adding technological measures to improve automation, including RFID tagging, OCR-based scanning, GPS-based positioning and mesh networking, critical processes – including asset movement scheduling and tracking – are still handled manually for the most part.

Groups of researchers at Georgia Tech are already looking at many different aspects of operations automation to reduce costs, improve efficiency and increase utilization. Since the overall operations of the Port are quite complex, involving many different actions spanning a variety of assets and vehicles, we will focus on a specific illustrative example – a

[5] `http://www.gaports.com/`

truck picking up a container unloaded at the Port. A truck's pickup flow can be monitored from the time a truck arrives and is inspected at one of the Port's truck entry lanes to the time a truck leaves with cargo (and undergoes an outgoing inspection). Since unexpected problems or coordination failures may delay a truck's unimpeded flow from entry to exit and lead to cascading delays, such tracking would allow the system to detect and potentially remediate such issues (or at least alert staff). Employees already look out for these problems, but automated tracking has the potential for wider coverage and lower detection latency.

Activity tracking from fundamental sensor streams such as video is an active area of research. Researchers have demonstrated techniques for inferring anomalies in videos of delivery vehicle loading and unloading [98] and for tracking targets in sparse camera networks [196]. For our evaluation purposes, we will assume the existence of appropriate fundamental analysis techniques. Since our previous two application scenarios already demonstrate computationally significant feature detectors running on fundamental streams, here we will focus on the interplay between higher-level anomaly detection and the high-level features of *temporal streams* – specifically its time-based data abstraction and support for historical data.

Our example will assume that some lower-level analysis is feeding data into a higher level inferencing mechanism to detect anomalies – we will model the higher level inferencing using a *rule engine*. If an anomaly is found, an alert is signaled. The system must also use historical data from appropriate timeframes to do exploratory postmortem analysis involving interactive feedback from tracking personnel. A system might prefer reactive analysis to exhaustive routine monitoring if the computational cost of constant routine analysis is too high or if current techniques are not yet capable or accurate enough for full

automation. In the latter case, a partially automated strategy can be used where the system performs analysis guided with interactive feedback.

## *8.2 Implementation*

In this section, we will describe how each application scenarios is implemented using *temporal streams*. We will describe the components of each implementation in detail, augmenting our discussing with specific code examples. The code examples presented here are excerpted and somewhat stylized for illustrative clarity (primarily in the form of simplified error checking and elided boilerplate); nonetheless, all of the examples come from the actual application implementations. Following our three application implementation subsections, Section 8.2.4 compares the structure and key details of the different application scenarios.

For the vision-based video feature detectors used in the airport and traffic scenarios, we primarily use the open source computer vision library OpenCV 1.1 [14]. JPEG encoding or decoding uses the standard `libjpeg6b` library. The airport and traffic applications are constructed in C, using the C-based *temporal streams* runtime. The port scenario uses the Java-based *temporal streams* runtime and the Drools [8] Business Rules Engine version 4.0.7.

### 8.2.1 Airport Surveillance

The airport surveillance application scenario consists of seven basic components: 1) "agents" hosting video and sensor channels, 2) video data producers, 3) sensor data producers, video feature detectors – 4) face detection and 5) optical flow, 6) historical query generators, and 7) feature aggregators. Figure 44 depicts the dataflow between components. We derive

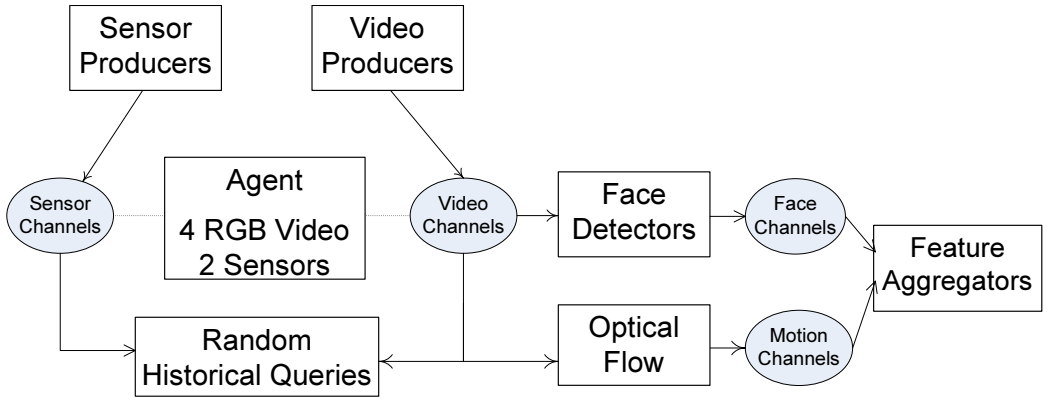



**Figure 44:** Airport Surveillance – Components & Dataflow

the structure of our airport surveillance application from ASAP [180]. ASAP is a situational awareness system for applications such as video-based surveillance, including airport surveillance.

**Agent:** An agent is a component hosting multiple persistent video and sensor data channels. Each agent will create locally hosted video and sensor channels and add a pickling handler on each video channel. The pickling handler will take the raw RGB video frames and transform them into JPEG images when data is moved to persistent storage.

**Video Data Producer:** The video data producers are quite simple; each producer decodes MJPEG video files to RGB video frames and places each frame into a video channel using the default timestamp (i.e., *now* at the time the *put* call is executed). This component's structure follows the basic data producer example depicted in Figure 7. Each video frame is 225KiB and the frame rate is approximately 29fps.[6]

**Sensor Data Producer:** The sensor data producers are like the video data producers, except they produce smaller, 1024-byte items representing sensor data readings. The data

[6]The video is captured in a custom format, hence the unusual framerate.

```
int main(int argc, char **argv) {
  rt_sys_handle_t *rtsh;
  conn_endpt_t inchan, outchan;
  cmdargs_t args;

  // Parse command-line args
  parse_args(&args);

  // Connect to front end and contact super-nodes.
  rtsh = INIT_RT(argv[1]);

  // Get channel descriptor.
  inchan = ADD_CHAN(rtsh, args.input_name);
  outchan = ADD_CHAN(rtsh, args.output_name);

  // Run the transform function
  transform(rtsh, &args, &inchan, &outchan);

  return EXIT_SUCCESS;
}
```

**Figure 45:** Simple Unary Feature Detector – `main`

production rate is 15 items per second.

**Face Detector:** The face detector component takes input video frames and isolates image regions containing potential faces using OpenCV's face detector. The OpenCV [14] face detection functionality is based on Viola and Jones's object-detection techniques [198], using Rainer Lienhart's stump-based trained 20x20 frontal face detector Haar cascade.[7] Figure 48 shows an example of OpenCV's face detector running on live video. The structure of this component is a straightforward unary feature detector as depicted in Figure 45 and Figure 46. The output of the face detector is a 128-byte vector of image coordinates

[7] OpenCV's `haarcascade_frontalface_alt.xml`.

```
void transform(rt_sys_handle_t *rtsh, cmdargs_t *args,
               conn_endpt_t in_ch, conn_endpt_t out_ch) {
  // Create local copy of in_ch to in
  // Set up buffers, local vars
  // ...

  while(!done) {
    // Get one item
    item->status = CONN_GET_1_I(in, &low, &up, item);

    // Successful get
    if(item->status == 0 && item->buf_len >= 1) {
      low = TIME_ADD(&item->ts, &one_micro);

      // Process item, result in buf2

      rval = CONN_PUT(out_ch, buf2, buf2_sz, &item->ts);
    }

  }
  // ...
}
```

**Figure 46:** Unary Feature Detector – `transform`

corresponding to $(x, y)$ for potential faces bounds. This component uses a local replica of the input video channel so the fetching of video frames is partially overlapped with the feature detector computation.

**Optical Flow:** The optical flow feature detector estimates motion from the raw video frames. The processing consists of two steps: 1) transforming each RGB video frame into grayscale and 2) performing the optical flow calculation. We use OpenCV's implementation of Horn and Schunck's optical flow algorithm [106] and a simple hand-coded

```
decode_vid_hdr(item->buf, &height, &width, &bpp, &c);

// make grayscale
rgb_to_grayscale(buf3, DATA_OFFSET(item->buf),
                 width, height, bpp);

// halve framerate
if(i & 1 == 1) {
  goto skip;
}

// do OpenCV optical flow
n = detect_optflow1(buf3, width, height, buf2);

// ... send results in buf2 ...
```

**Figure 47:** Unary Feature Detector – Optical Flow Processing Example

grayscale converter.[8] Like face detection, the structure of the optical flow component is a unary feature detector as depicted in Figure 45 and Figure 46. Figure 47 presents the specific processing operations for our optical flow component (these would occur in Figure 46 where the comment indicates that the data item is processed). This component uses a local replica of the input video channel so the fetching of video frames is partially overlapped with the feature detector computation.

The calculation of optical flow requires two video frames (as motion is estimated from inter-frame differences), so we buffer the previous video frame to use it with each newly fetched video frame. Note that we perform the grayscale conversion on all images, but only run the optical flow calculation on every other image (hence the "halve framerate"

[8]The RGB to grayscale conversion uses color weights of 0.299 for red, 0.587 for green and 0.114 for blue (weights specified in ITU-R BT.601-6 "Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios").

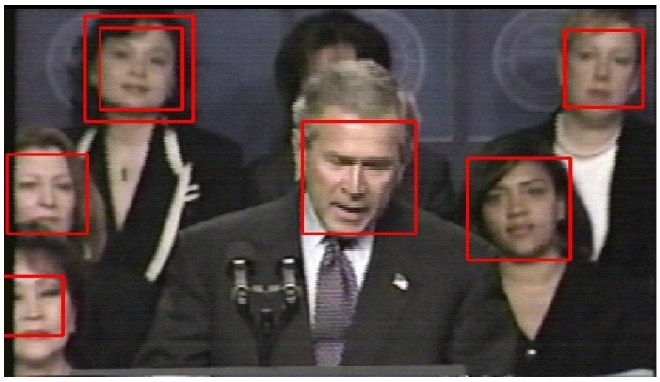


**Figure 48:** Airport Surveillance – Face Detection Example

comment in Figure 47). This is an artifact of our testing environment – the optical flow calculation is relatively expensive and we cannot perform full-framerate calculation on our target hardware (see Section 8.3.1). The output of the optical flow feature detector is a 128-byte digest of motion information.

```
while(true) {
  // Generate a random time interval based on rp's params
  random_gen(&lower, &upper, percentage, &rp);

  // Get rp.n items
  rval = conn_get_n(chan, &lower, &upper,
                    bufs, bufsizes, rp.n, metadata);

  ...
}
```

**Figure 49:** Airport Surveillance – Historical Query Generator

**Historical Query Generator:** A historical query generator performs periodic *get* operations on both live and historical data designed to simulate human interest on a given input channel (either a video or sensor channel). There is one query generator for each raw data

producer (i.e., one generator for each video channel and one for each sensor channel). Periodic queries follow a specific probability distribution, and the choice between fetching historical or live data is determined by a unbiased coin flip. The choice of what historical data to fetch then follows a power-law distribution over the range of archived data. The power-law distribution roughly approximates the application-specific situation where most video captured will be uninteresting with a few periodic events of high interest. Figure 49 shows how the query generator retrieves both live and historical data.

**Feature Aggregator:** The feature aggregator gathers data from all channels corresponding to a particular class of feature detector. For example, if there are five optical flow outputs in an application, the feature aggregator will consume data from all five optical flow output channels. In our airport surveillance application, we use two feature aggregators – one for optical flow, and one for face detection. The feature aggregators represent agents using feature detector outputs to perform higher-level inferencing; in our implementation, they are just used to measure end-to-end processing latencies.

### 8.2.2 Traffic Monitoring

For our traffic monitoring application scenario, we implement five major components: 1) video data producers, 2) feature aggregators, and three video feature detectors, 3) background maintenance, 4) foreground separation, 5) camera change detection. Figure 50 depicts the dataflow between these components, including the channels present.

**Video Data Producer:** The video data producers are simple and straightforward; each producer decodes specially-produced MJPEG video files to RGB video frames and places each frame into a video channel using the default timestamp (i.e., *now* at the time the *put* call is executed). The structure follows the basic data producer example depicted in

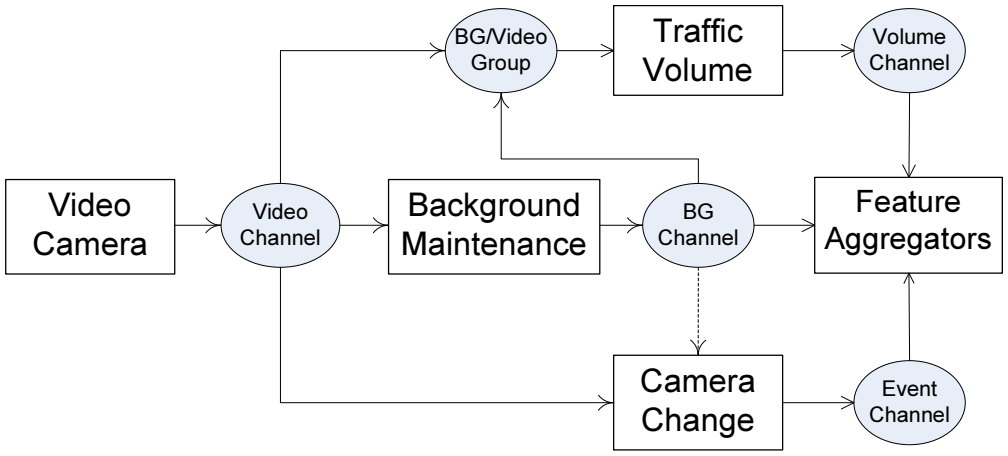



**Figure 50:** Traffic Monitoring – Components & Dataflow

Figure 7. Each video frame is 225KiB and the frame rate is 30fps.

**Background Maintenance:** The background maintenance feature detector component creates a simple background model from raw video frames. The component consumes video frames and maintains a simple moving average like OpenCV's `cvRunningAvg` function with $\alpha = 0.1$. Figure 51 shows an example raw video frame and a background model corresponding to that video frame. The structure of the background maintenance component is a straightforward unary feature detector as depicted in Figure 45 and Figure 46. The background maintenance component uses a local replica of the input video channel so the fetching of video frames is partially overlapped with the background model computation. The background model output data is the same size as a source video frame (225KiB).

**Foreground Separation / Traffic Density:** The foreground separation feature detector component attempts to estimate traffic density by separating the moving foreground of the raw video image from the static background. This process involves image differencing using two inputs – the background model and the raw video frames. This component is a

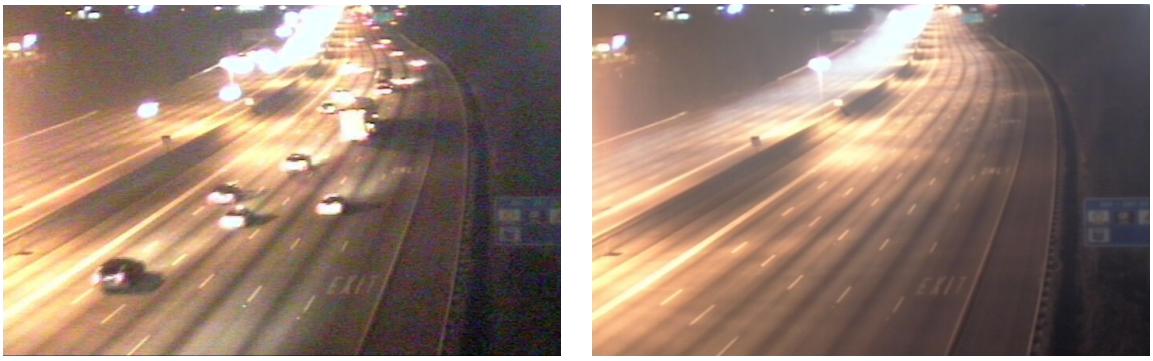


**Figure 51:** Traffic Monitoring – Background Maintenance Example

binary feature detector using inputs from both the background maintenance and the video producer; since it needs to use up-to-date background models, the two input channels are put into a channel group for virtual synchronization. The structure is depicted in Figures 52, 53 and 54. Figure 56 shows an example raw video frame and a foreground corresponding to that video frame. The output of this feature detector is a 128 byte data item containing estimated traffic density information.

**Camera Change Detection:** The camera change feature detector finds abrupt changes in camera positioning by using histograms. It is also a binary feature detector, potentially using inputs from both the original video data and the background model. Unlike traffic density estimation, however, the camera change detector only uses input from both channels occasionally – so the feature detector input is either unary or binary depending on dynamic application conditions. Normally, the system performs histogram calculations on raw video data, but upon detecting a sudden change, it retrieves a range of background model data to analyze the cause and nature of the change. The structure of this component is depicted in Figures 52, 53 and 55. The output of this feature detector is a 128 byte data item containing estimated camera change information.

```
// ...
// Parse command-line args
// Connect to front end and contact super-nodes.

// Get channel descriptor.
inchan = ADD_CHAN(rtsh, args.input_name);
inchan2 = ADD_CHAN(rtsh, args.input_name2);
outchan = ADD_CHAN(rtsh, args.output_name);

// Create channel group
int ch_nums[2] = {inchan.chan_num, inchan2.chan_num};
MAKE_GROUP(rtsh, ch_nums, gr_nums, 2, 1);

// ... Create background get thread ...
// Run the transform function
```

**Figure 52:** Binary Feature Detector – `main`

**Feature Aggregator:** The feature aggregator gathers data from a limited number of similar feature detectors. For example, in a run of the traffic monitoring application, one feature aggregator will consume data from four traffic density output channels. In our application, we run feature aggregators for both camera change detection and traffic density. As with the previous application scenario, the feature aggregators represent agents using feature detector outputs to perform higher-level inferencing but we use them to measure end-to-end processing latencies in our system.

### 8.2.3 Port Asset Tracking

Since port asset tracking involves a large space of potential events and behaviors, we will focus on an small illustrative subset of a full conceptual application. In particular, we will focus on a truck's flow through the port to pick up cargo and depart; this application cross-section highlights the interplay between higher-level inferencing and time-oriented data

```
// ...
while(!done) {
  // Get one item
  item->status = CONN_GET_1_I(in, &low, &up,
                              item, gr_nums[1]);

  // Successful get
  if(item->status == 0 && item->buf_len >= 1) {
    low = TIME_ADD(&item->ts, &one_micro);

    ATOMIC_PUT(&globbg, item->buf);

    // ...
  }
}
```

**Figure 53:** Binary Feature Detector – Background Fetching Thread

access. For the purposes of our application, we will assume that a truck flows from port entry to port exit and passes various checkpoints along the way.

In our implementation, we use a declarative business rules engine called Drools [8] to interpret high-level events as they occur. We take for granted the ability to produce event information since the previous two application scenarios already involve the transformation of fundamental component streams to feature streams. Drools is Java-based, so we use the Java *temporal streams* runtime. The example code presented here is in Drools 4.x format. Each named rule is of the simple form depicted in Figure 57. As the figure shows, each rule consists of a `when` clause followed by a `then` clause. The `when` clause is set of declarative pattern-matching rules specifying conditions concerning facts in the rule engine's *working memory*. When facts are inserted into working memory, the various rules' `when` clauses are evaluated; if a rule is true, meaning all of the `when` conditions are satisfied, the `then`

```
decode_vid_hdr(item->buf, &height, &width, &bpp, &c);

// grab current background
ATOMIC_GRAB(&databuf, &globbg);
cvSetImageData(&bgImg, DATA_OFFSET(databuf), width * 3);

// do foreground separation
res = detect_fgdata1(DATA_OFFSET(item->buf), &bgImg,
                     &tmpIn, width, height, buf2);

// ... send results in buf2 ...
```

**Figure 54:** Binary Feature Detector – Foreground Separation Example

actions are executed. These actions may modify working memory by adding or removing facts. Actions are specified in a Java-like dynamic expression language called MVEL.[9]

The first application rule is the simplest: Figure 58. This rule just logs whenever a truck enters the port. The rule in Figure 58 fires when an object of type `TruckEnterEvent` is newly inserted into working memory. The `when` clause asserts that such an object exists and binds it to the `$tee` variable so the event object causing the rule to execute can be used in the corresponding action. This rule will fire each time a new `TruckEnterEvent` is generated.

When a truck leaves, the corresponding "Truck Leaving Port" rule listed in Figure 59 should execute. The rule triggers when a `TruckEnterEvent` and `TruckLeaveEvent` both exist with matching `id` fields – this means that the same truck has previously entered the port and is now leaving. The rule's actions calculate the number of seconds between entering and leaving and then log the event. After logging, the rule action removes both the enter

[9] `http://mvel.codehaus.org/`

```
decode_vid_hdr(item->buf, &height, &width, &bpp, &c);

// histogram scene change
res = detect_schist1(DATA_OFFSET(item->buf),
                     &lastHist, width, height,
                     corr_hist, ncorr_hist,
                     &curr_start, 0.5);
...

if(res > threshold) {
  upper = NOW();

  // fetch BG images from bg_chan to confirm
  lower = TIME_SUB(item->ts, TIME10MS);
  CONN_GET_N(bg_chan, &lower, &upper,
             bufs, bufsizes, 10,
             NULL, gr_nums[1]);
  // ...
```

**Figure 55:** Binary Feature Detector – Camera Change Example

and leave events from working memory and inserts a new `TruckCycle` fact to represent a complete trip from entry to exit. The `TruckCycle` is subject to further processing by other rules.

Now that we have demonstrated basic rules for normal scenarios, we show a simple anomaly detection rule in Figure 60. The "Phantom Truck Leaving Port" rule finds a `TruckLeaveEvent` where no corresponding `TruckEnterEvent` exists. When the "Phantom Truck" rule is triggered, the system logs the anomaly and attempts to diagnose it. Figure 61 shows the actions taken. The system attempts to find the missed automatic truck entry event by involving the human operator. First, the system finds the current average truck cycle time and uses that to generate a guess for the "phantom" truck's arrival time.

**Figure 56:** Traffic Monitoring – Foreground Separation Example

```
RULE "Syntax Example"
  WHEN
       [condition 1]
       [condition 2]
       ...
  THEN
       [action 1]
       [action 2]
       ...
END
```

**Figure 57:** Port Asset Tracking – Drools Syntax Example

It then fetches video from the camera monitoring the port's truck entry point; here we use *temporal streams*, fetching video from the corresponding channel based on a historical time interval bracketing ±10% of the system's estimated entry time. The operator watches this video and either marks the truck's arrival or modifies the search parameters (for example, based on knowledge of recent backups or other reasons why the entry time would be different from the estimated time). Eventually, the truck entry is found and the corresponding `TruckEnterEvent` is generated or the operator gives up and the system removes the orphaned `TruckLeaveEvent` from working memory.

The rule in Figure 62 shows another anomalous event: a truck leaving the port late. If

```
RULE "Truck Entering Port"
  WHEN
    $tee : TruckEnterEvent()
  THEN
    log($tee)
END
```

**Figure 58:** Port Asset Tracking – Truck Entering Port

```
RULE "Truck Leaving Port"
  WHEN
    $tle : TruckLeaveEvent($tsl: timestamp, $id1: id)
    $tee : TruckEnterEvent($tse: timestamp, id == $id1)
  THEN
    diff = ($tsl.getTime() - $tse.getTime()) / 1000)
    log("Truck %d leaving at %d secs", $id1, diff)
    retract($tle)
    retract($tee)
    insert(new TruckCycle($tee, $tle, diff))
END
```

**Figure 59:** Port Asset Tracking – Truck Leaving Port

```
RULE "Phantom Truck Leaving Port"
  WHEN
    $tle : TruckLeaveEvent($id1: id)
    not ( exists (TruckEnterEvent(id == $id1)) )
  THEN
    log("No enter event: phantom truck %d", $id1)
    // diagnose anomaly
    diagnosePhantom($tle)
END
```

**Figure 60:** Port Asset Tracking – Phantom Truck Leaving Port

```
// rule "Phantom Truck Leaving Port" actions
void diagnosePhantom(TruckLeaveEvent tle) {
  Long ts = tle.getTimestamp().getTime() / 1000;

  // based on estimated truck timing, find potential
  // video of entry
  Long tentry = ts - getAvgCycleTime();
  Long tdelta = getAvgCycleTime() / 10;
  boolean done = found = false;

  while(!done && !found) {
    // ± 10% around potential entry
    Item[] i = entryCam.getItems(tentry - tdelta,
                                 tentry + tdelta);

    // show human operator
    OpOutput oo = showOperator(i);

    // using operator feedback, continue searching
    tentry = oo.getTEntry();
    tdelta = oo.getTDelta();
    found = oo.isFound(), done = oo.isDone();
  }

  // ...

  // if we can't find anything, log failure and
  // remove truck leave from working memory
  if(!found) {
    log("Failed to find %s", tle);
    retract(tle);
    done = true;
  }
}
```

**Figure 61:** Port Asset Tracking – Phantom Truck Diagnostics

```
RULE "Truck Leaving Port Late"
  WHEN
    // >55 mins is late (time is in seconds)
    $tc: TruckCycle(time > (60*55))
  THEN
    log("Truck Left Late!")
    // diagnose anomaly
    diagnoseLateDeparture($tc)
END
```

**Figure 62:** Port Asset Tracking – Truck Leaving Port Late

we find a `TruckCycle` object in working memory where the corresponding time is greater than 55 minutes, we log the anomaly and attempt to find potential issues leading to the late truck departure. Figure 63 shows the diagnostic actions taken. The system uses the timing information corresponding to a truck's entry and departure to find out why the truck has been delayed. The system uses an iterative feedback process involving the human operator to isolate the earliest point at which the truck is delayed. This process is a binary search, starting with the estimated mid-point of the delivery. The system finds a video channel which would observe a truck at the projected point in the delivery process, looks up some historical video and shows it to the operator. The operator can guide the process by indicating that the truck is still on time at the point specified, meaning that the delay occurred later in the cycle, or the truck is already late, meaning the delay occurred earlier in the cycle. This process shows the use of *temporal streams*-provided historical data spanning multiple channels.

Finally, we show another anomalous event detection rule in Figure 64 – it detects when a truck misses a scheduled checkpoint. In the rather complex `when` clause, we find a `CheckptEvent` where no corresponding `CheckptEvent` exists to signify the immediately

```
// rule "Truck Leaving Port Late" actions
void diagnoseLateDeparture(TruckCycle tc) {
  Long len = tc.getTruckLeaveEventTime() -
             tc.getTruckEnterEventTime();
  Long tdelta = len / 10;

  // do a binary search for the truck delay; find
  // the first point where the truck starts running late

  // start at the middle of the journey
  Long ts = tc.getTruckEnterEventTime() + (len / 2);

  while(!done) {
    // get appropriate channel for time in delivery cycle
    chan = getVideoChannelByTime(tc, ts);

    // ...

    Item[] i = chan.getItems(ts - tdelta, ts + tdelta);

    // show operator video
    OpOutput oo = showOperator(i);

    // modify search based on feedback
  }

  // ...
}
```

**Figure 63:** Port Asset Tracking – Late Departure Diagnostics

preceding truck checkpoint (excluding the first, which has no predecessor). For example, if checkpoints 1, 2 and 4 are present, this rule would trigger with `$ce1` bound to the `CheckptEvent` with `point` = 4 (because no `CheckptEvent` exists with `point` = 3). The other two clauses of the rule collect all prior checkpoints into a list and find the most recent checkpoint present. In our example, `$prior` would contain `CheckptEvent`s corresponding to 1 and 2 and `$lp` would be equal to 2. Figure 65 shows the diagnostic actions taken: the system grabs video data from the time interval between two good checkpoints immediately bracketing the missing checkpoint. The operator can then use this information to determine the cause of the anomaly and take action (if required).

For a more slightly more advanced implementation, consider modifying the code in Figure 65 from lines 10 to 15. Instead of fetching and showing raw video, we could fetch data corresponding to the same time interval from a feature stream containing foreground objects isolated from the raw video. This feature stream data would be more sparse than the video data from which it is produced because it only contains moving objects. We could then use this data to match up potentially relevant foreground objects with corresponding video data using *temporal streams*'s timing information, thus narrowing down the amount of video the operator must watch to a subset of promising times of interest (with object motion). This could be extended to query multiple cameras covering a common geographical region, using the foreground feature stream data to aggregate intervals of interest in the corresponding fundamental camera video data.

### 8.2.4 Comparison

Although the three previously described application scenarios all have commonalities, they highlight different properties of live stream analysis applications; their diversity spans a

```
RULE "Truck Missed Checkpoint"
  WHEN
    // if we have a gap in checkpoints
    $ce1 : CheckptEvent($id1: id, $point: point > 1)
    not( exists ( CheckptEvent(id == $id1 &&
                               point == ($point-1)) ))
    // collect all prior checkpoints into a list
    $prior : ArrayList() from
             collect( CheckptEvent(id == $id1,
                                   $p2: point < $point) )
    // find the most recent checkpoint before missing
    $lp : Number() from
          accumulate( CheckptEvent(id == $id1,
                                   $p2: point < $point),
                      max($p2) )
  THEN
   log("Truck %d missing checkpoint #%d", $id1, $point-1)
   log("Last checkpoint #%d: %s", $lp, $prior)
   // diagnose anomaly
   diagnoseMissedCheckpoint($ce1, $prior, $lp)
END
```

**Figure 64:** Port Asset Tracking – Truck Missed Checkpoint

```
// rule "Truck Missed Checkpoint" actions
void diagnoseMissedCheckpoint(CheckptEvent ce,
                              ArrayList<CheckptEvent> lst,
                              Long lastPointId) {
  HashMap<Long, CheckptEvent> map = pointListToMap(lst);
  CheckptEvent last = map.get(lastPointId);

  // ...

  Channel c = getVideoChanByCheckpt(lastPointId+1);

  // get data between two good checkpoints
  Item[] i = c.getItems(last.getTimestamp(),
                        ce.getTimestamp());

  // use video for diagnosis

  // ...

}
```

**Figure 65:** Port Asset Tracking – Missed Checkpoint Diagnostics

range of potential application requirements. The airport surveillance scenario uses heavy-weight unary feature detectors with a fairly regular graph structure. It also makes use of historical data and *temporal streams*'s persistence mechanism and pickling functions. The traffic monitoring scenario highlights multi-stage fundamental feature detector chains – in particular, both the traffic density estimation and camera change detection use inputs from the background maintenance component. These two second-level feature detectors also demonstrate multiple inputs; the traffic density component uses a channel group to dynamically synchronize its two inputs – the raw video data and a corresponding background model. On the other hand, the camera change detection component dynamically changes between consuming input data from one or two channels. When a potential camera change is detected, the system uses extra background model data to confirm. The number of inputs is dynamically determined by properties of the data itself – properties extracted by the feature detector.

Finally, the port asset tracking scenario demonstrates the interplay between higher-level inferencing in stream analysis applications and the programming model's time-based data access plus the ability to access arbitrary intervals of stream content potentially spanning historical data. Since the airport and traffic scenarios involve the transformation of fundamental component streams to feature streams, we do not focus on those aspects in the port asset tracking scenario; instead we assume those capabilities as a given. The declarative rules used to detect and respond to anomalous conditions "close the loop" between higher-level event behavior and lower-level data. The system attempts to diagnose and remediate problems by fetching relevant data from corresponding time intervals and allowing human operators to quickly troubleshoot anomalies. This diagnostic process involves both automatic analysis and human judgement; although many kinds of analysis can be

entirely automated, involving personnel in quick directed feedback increases the overall quality of results. This kind of synergistic semi-automated monitoring will be common in cyber-physical systems as long as humans are significantly better at some kinds of tasks than current state-of-the-art computational techniques.

## *8.3 Experimental Evaluation & Analysis*

In this section we will provide a quantitative experimental evaluation and analysis of both the airport surveillance application (Section 8.3.1) and the traffic monitoring application (Section 8.3.2). Each application section will briefly recap the components outlined in Section 8.2 for suitable, experiment-relevant context; in addition, we describe the experiment-specific configurations of these components, including the connection topology, the assignment of components to physical nodes and the data inputs and parameters used. Since both experiment series are run on the same cluster resources, we will first describe some common hardware and software configuration details as well as baseline communication measurements.

**Common Experimental Setup:** Our quantitative application experiments are run on a cluster of dual-processor 64-bit Linux nodes. Each node has two Pentium 4-based (NetBurst) "Nocona" Xeon 3.2GHz processors with 1MiB of L2 cache, 800MHz FSB, 6GiB of RAM, and IP over Infiniband networking (4x SDR). Under moderate load, observed ICMP round trip latency is about 100 $\mu$s with a 256 byte payload. The typical observed peak IPoIB throughput is about 2.5Gbits (see below for more in-depth and application-relevant network measurements). For the first set of experiments (the airport scenario), the nodes run RHEL 4u6, kernel 2.6.9-67.0.1.ELsmp (64-bit). Due to an upgrade, the nodes are running RHEL 4u7, kernel 2.6.9-78.0.13.ELsmp (also 64-bit) for the second set of experiments

(the traffic scenario). All binaries are built with gcc 4.1.2 with `-g` and `-O2` (debugging and optimization). Persistent channels in the airport scenario use the `fs1` storage backend, writing to `/tmp` on the channel hosting node – `/tmp` resides on an ext3 filesystem on a Seagate ST373207LC 10k RPM SCSI drive. For uniformity, we use the C *temporal streams* runtime with the first binary wire protocol version (see Chapter 6 for runtime details) for all quantitative experiments.

**Baseline Network Metrics:** In order to get baseline measurements for the communications performance of the cluster nodes, we use the netperf tool [112]. netperf provides a "TCP Round Robin" mode (`TCP_RR`) which measures the throughput of a TCP request-response sequence over an established, persistent connection. `TCP_RR` mode measures the achievable throughput when performing a common request-response communication pattern – this pattern is identical to our implementation's communication protocol when fetching channel data (see Section 6.6.2), so it should provide a good frame of reference for all of our quantitative experiments. In order to model our protocol, we use a request size of 64 bytes, matching a simulated *get* request. We set the response size to 230,492 bytes, as this matches our experimental video data response size: the 24-bit RGB 320x240 video frame is 230,400 bytes (225KiB) with an additional 8-bytes of video metadata, 20-bytes of channel item metadata and the 64-byte *get* response header. Using these parameters, we run netperf from several pairs of unloaded hosts ten times. One representative result is presented in Figure 66.

All of the tests reported between about 1170-1176 transactions per second, and the netperf process incurred about 55-70% of a single CPU to sustain peak `TCP_RR` throughput. These transaction rates imply a best-case channel *get* operation for a single video frame

```
$ netperf -p 12865 -t TCP_RR -H rohani16 -4 -l 30 -i 10 -- -D -r 64,230492

TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
rohani16.cc.gatech.edu (130.207.115.136) port 0 AF_INET :
+/-2.5% @ 99% conf.  : nodelay
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  87380  64       230492  30.01    1174.43
16384  87380
```

**Figure 66:** Example `netperf` results on test cluster

should take 0.85-0.86 milliseconds on an unloaded (otherwise idle) system. 1176 transactions per second of 225KiB video frames gives an effective maximum payload data rate of about 258MiB per second.[10]

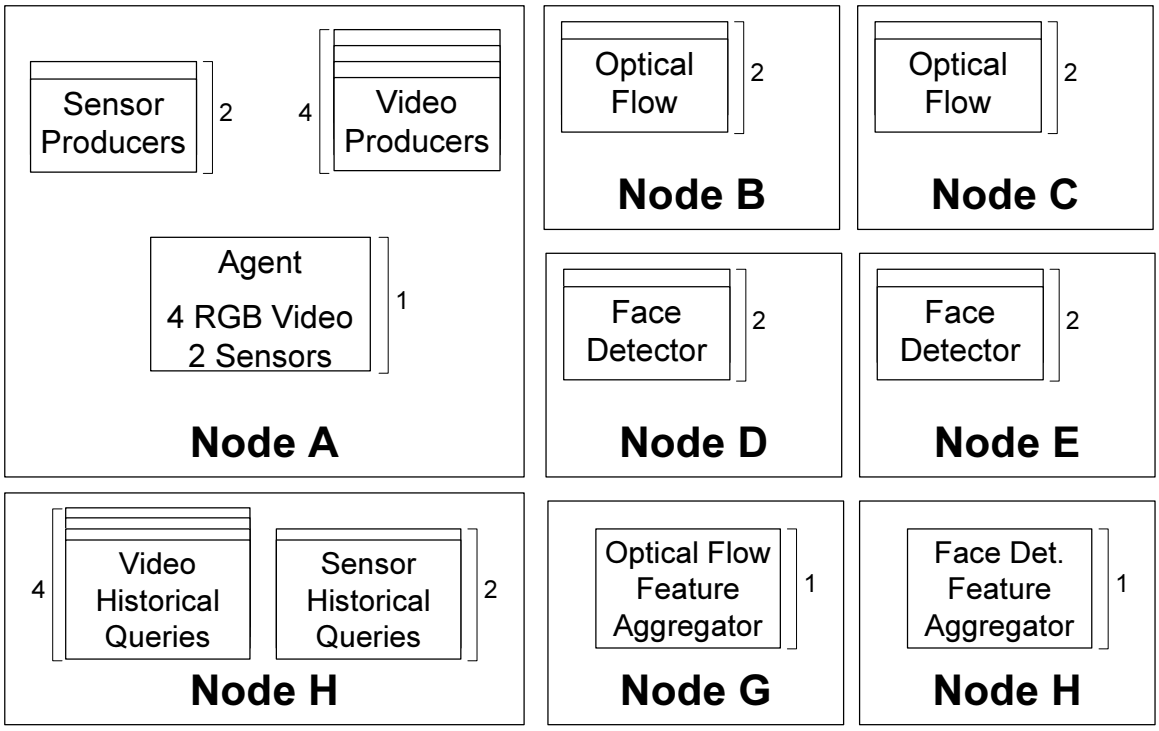
### 8.3.1 Airport Surveillance

**Components Recap:** Our airport surveillance application on *temporal streams* consists of seven parts: 1) the "agents" hosting video and sensor channels, 2) video data producers, 3) sensor data producers, video feature detectors – 4) face detection and 5) optical flow, 6) random query generators, and 7) feature aggregators. Figure 44 depicts the dataflow between components. Each agent hosts some number of persistent video and sensor channels. The video data is 225KiB RGB video frames at ~29fps, transformed to JPEG format using a pickling handler. The sensor data is random 1024 byte samples produced at 15fps. Video producers generate the RGB video frames by decoding MJPEG 3-4 minute compressed video files captured from TV and playing them back on a loop. The feature detectors get

[10](1176 ops. per sec × 225 KiB per op.) ÷ 1024 KiB/MiB = 258.3984375 MiB/sec

video frames one at a time from the channels and run either face detection or optical flow analysis on each frame. They overlap computation and communication by simply getting video data to process from a local replica of the video channel. The optical flow process first converts each video frame to grayscale but only performs the subsequent optical flow computation on every other frame (the CPUs limited our ability to do full frame-rate optical flow). Halving the frame-rate keeps a CPU busy approximately 65-70% of the time. Each feature detector outputs a small 128-byte digest of the results into a channel. The random query components generate random historical and live data queries on the video and sensor data with a specific probability distribution. Finally, there is a single feature aggregator for each feature detector type (face detector or optical flow); each aggregator gets the results from all feature detectors of the given type (corresponding to all video channels) and calculates the latency of processing video frames. All components process data in order and do not drop frames.

**Topology:** In our setup, we host four video channels and two sensor channels per agent, with one agent per cluster node. The four video producers and two sensor data producers corresponding to the agent are also collocated on the same node, although they are logically separate processes. This node will be decoding 4 MJPEG video streams to produce RGB video frames, encoding 4 MJPEG video streams from the same RGB frames for pickling handlers and committing all six data streams to disk. It is also responsible for serving video content to eight feature detectors (four of each type) and handling live and historical queries from the random query generators. The rest of the pieces run on independent nodes in different groupings. The feature detectors run two per node and host their own output channels locally. The random query generators run six per node (four video, two sensor) and both measurement aggregators run on separate nodes. For our experiments, we use two




This figure depicts the node assignment of components with four video channels and two sensor channels. This setup would be replicated for every added four video channels (and two sensor channels) except for the feature aggregators; a single instance of each type of aggregator exists regardless of the number of total video channels.

**Figure 67:** Airport Surveillance – Topology

agents and eight video streams total. We use a total of 14 cluster nodes. Table 8 summarizes this setup, and Figure 67 graphically depicts the component to node assignments for a four video channel subset of the experiment.[11]

**Workload Characteristics:** We perform five runs of each experiment, each normal run lasting six minutes and involving about 10,500 frames of video for each channel. Each

[11]To extrapolate to the eight video channels used in this experiment, the entirety of Figure 67 would be replicated except for the two aggregation components.

**Table 8:** Airport Surveillance experiment

| Component | Configuration | Total | +Nodes |
|---|---|---|---|
| Agent | 1 per node, hosts 4 vid. / 2 sensor | 2 | 2 |
| Producers | 6 per ASAP node, one per stream | 12 | - |
| Historical Query | 6 per dedicated node | 12 | 2 |
| Face Detection | 2 per dedicated node | 8 | 4 |
| Optical Flow | 2 per dedicated node | 8 | 4 |
| Face Aggregator | 1 per dedicated node | 1 | 1 |
| Optical Flow Aggregator | 1 per dedicated node | 1 | 1 |

The "+Nodes" column specifies the additional nodes required by each column.

query generator makes a video query every 100ms requesting a live or historical frame with equal probability. The historical frames' timestamps are chosen based on a probability distribution that is roughly Zipfian[12] (we use a power-law distribution to approximate a situation where most video captured will be uninteresting with a few periodic events of high interest). The standard configuration has all streams converted to MJPEG before being stored to disk. The 1RGB configuration has one stream per agent (two total streams) stored to disk without compression to increase the size of the historical data-set. Similarly, the 2RGB configuration has two streams per agent stored without compression. Due to the large amount of RAM on each node, the set of files comprising all historical streams can fit in buffer cache easily on the shorter runs. Consequently, we also run some significantly longer experiments to ensure that the historical data set size is large enough to ensure that all requests cannot be serviced from RAM. All of the longer experiments have one RGB stream per agent.

[12]It's not a true Zipf distribution since $N$ changes over time.

**Feature Detector Results:** Figure 68 shows the feature detector latencies (in milliseconds) of several different configurations (and Table 9 presents the raw data): the first two columns are the processing latency measurements at the face detector and optical flow feature detectors. The Agg columns show the measured latency at the aggregators. In both cases, the latency is calculated using the timestamp of the original video frame. The aggregators include another network hop since they consume the feature detection output data stream; in addition, the aggregators get the newest item from all feature detectors of a given type in sequence rather than concurrently, and each get call can potentially block. Consequently, the feature aggregators' latencies (and standard deviation) increase with the number of streams they are consuming.[13] Having a separate thread handle each feature stream independently would alleviate this, but the straightforward sequential implementation is entirely adequate for ASAP and the performance is still quite good. Figure 69 shows a conceptual code example of the sequential feature aggregation consumer type we are using, while Figure 70 shows an example where all feature channels are measured concurrently.

Face detection is less expensive than the optical flow calculation, so the latencies are as expected. The baseline costs for the standalone feature detectors run on the same datasets on an unloaded node are shown as "Standalone." The face detection standard deviation

[13]For example, consider consuming items from three feature streams (*A*, *B*, and *C*) in sequence. If new items from feature streams *A* and *C* are ready immediately, but *B* takes a bit longer, and the feature aggregator reads the channels in alphabetical order, the *get* operation on *B* will block until *B* is ready. This means that the timing for *C* will include *B*'s delay, potentially over-counting. See Figure 69 – we use *now* measured after each channel *get* operation. If instead, we only measured *now* once at the beginning of each group of streams (e.g., after a *get* from *A*, the first stream), we risk under-counting the delay (since the end time is fixed for *B* and *C* after *A*'s read, so arbitrary delays by *B* or *C* are not counted). We prefer over-counting to potential under-counting. In practice, this is not significant in our application because the feature streams keep well in step. In addition, a real application's feature aggregation components would be performing analysis and not simply timing system latency; in this case, a developer would likely use the simplest method that maintained acceptable performance.

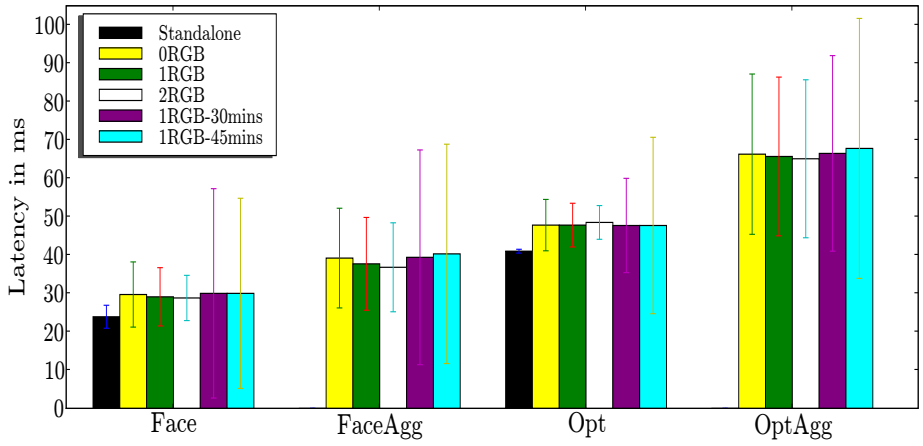



**Figure 68:** Airport Surveillance – Component latency in ms.

**Table 9:** Raw Data: Airport Surveillance – Component latency in ms.

| Config | Face | Optical Flow | FaceAgg | OptAgg |
|---|---|---|---|---|
| Standalone | 23.9 (3.0) | 41.0 (0.5) | - | - |
| 0RGB | 29.7 (8.5) | 47.8 (6.7) | 39.2 (13.0) | 66.3 (20.9) |
| 1RGB | 29.1 (7.6) | 47.8 (5.7) | 37.7 (12.1) | 65.7 (20.7) |
| 2RGB | 28.8 (5.9) | 48.5 (4.4) | 36.8 (11.6) | 65.1 (20.6) |
| 30mins | 30.0 (27.3) | 47.7 (12.3) | 39.4 (28.0) | 66.5 (25.5) |
| 45mins | 30.0 (24.8) | 47.7 (23.0) | 40.3 (28.6) | 67.8 (33.9) |

```
// in main
consume(chans);

...

// in consume
while(true) {
  for(int j=0; j < ntotal; j++) {
    upper = v_now();

    // Get one item
    item->status = conn_get_1_i(chans+j, lower+j,
                                &upper, item);

    // Successful get?
    if(item->status != 0)
      continue;

    ts[j] = item.ts;

    nowt32 = now();
    if(timercmp(&nowt32, (ts+j), >))
      timersub(&nowt32, (ts+j), (diff+j));
    else
      diff[j] = (struct timeval){0, 0};
 ...
```

**Figure 69:** Sequential Aggregator Example

```
// in main
for(int i=0; i < n_chans; i++) {
  pthread_create(thrs+i, NULL, consume, chans+i);
}

...

// in consume
while(true) {
  up = v_now();

  // Get one item
  item->status = conn_get_1_i(chan, &low, &up, item);

  // Successful get?
  if(item->status != 0)
    continue;

  ts = item.ts;

  nowt32 = now();
  if(timercmp(&nowt32, &ts, >))
    timersub(&nowt32, &ts, &diff);
  else
    diff = (struct timeval){0, 0};
 ...
```

**Figure 70:** Concurrent Aggregator Example

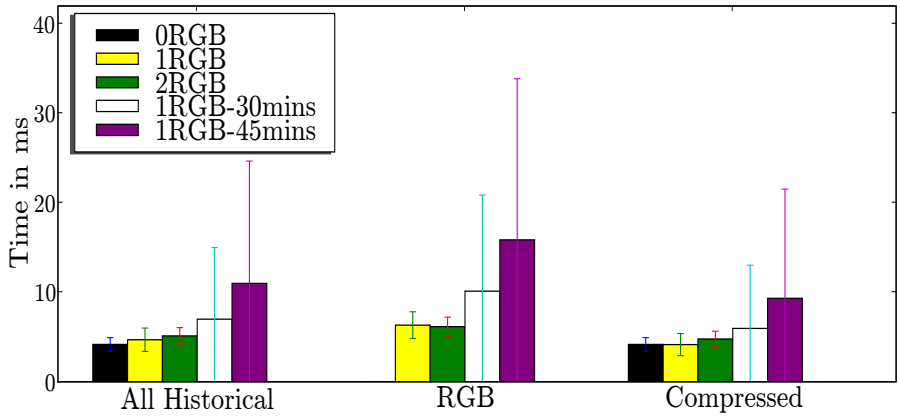



**Figure 71:** Airport Surveillance – Query time in ms.

is slightly higher because the face detection operation takes a varying amount of time depending on how many potential faces are present in an image, while the optical flow is only dependent on the image resolution. The deviation drops slightly going from *n*RGB to *n*+1RGB because the processing load decreases slightly with the removal of JPEG encoding pickling handlers. The variance on all components increases on the longer runs because of the effect of historical queries that cannot fit into RAM.

**Historical Query Results:** Figure 71 shows the average time to make a random historical query (in milliseconds) for video streams (and Table 10 presents the raw data). We separate the RGB and the compressed streams to show the effect of larger historical data sets. We can see that the average query time and variance grows as the amount of historical data grows – since the RAM size is constant, our locality gets worse as the total dataset grows. The compressed streams' latencies are affected too (but not as severely) because the same node and disk are used to host both types of streams.

**Other Experiments and Measurements:** We also ran the experimental configurations

**Table 10:** Raw Data: Airport Surveillance – Query time in ms.

| Config | All Historical | RGB | Others |
|---|---|---|---|
| 0RGB | 4.18 (0.75) | - | - |
| 1RGB | 4.70 (1.31) | 6.33 (1.49) | 4.16 (1.24) |
| 2RGB | 5.13 (0.92) | 6.17 (1.06) | 4.79 (0.87) |
| 1RGB-30mins | 7.0 (8.0) | 10.13 (10.73) | 5.97 (7.07) |
| 1RGB-45mins | 11.0 (13.67) | 15.87 (18.0) | 9.33 (12.20) |

with frame dropping by the feature detectors.[14] It negligibly lowered the latencies observed at the feature detector and aggregation components, although not significantly. The differences were consistent over various runs but about an order of magnitude smaller than one standard deviation; this is because the system rarely fell behind so dropping was also rare. Ultimately load-shedding in some form is essential to prevent overload situations where the system continues to fall further behind. Frame-dropping is one form of load shedding, but the decision on what form of load shedding to use depends on the application in question; some applications may prefer to lower the fidelity of analysis while continuing to process all data. In all cases, load shedding should reduce the components' average latencies and variances (by smoothing out peaks) if the system falls behind.

Table 11 lists the data sizes for the different application components and the system-imposed overhead in bytes. The per-item metadata size is a fixed 20 bytes and the per-response header overhead is 64 bytes. The worst-case total overhead is 84 bytes per item, and the per-item amortized total overhead can be less when multiple items are fetched within a single *get* request. The fixed overhead is negligible with medium to large data items in channels. For small items, the overhead is proportionally large but the overall data

[14] In other words, feature detectors would not process every video frame – if they fall behind they will skip items and always process the newest data available; see Section 7.1 for a more thorough description of "frame dropping."

size is still small, so it doesn't become significant unless there is an extremely large number of tiny data items. *Temporal streams* is not designed for application spaces where streams have a very large number of tiny items, but variable length metadata encoding in the second wire protocol revision cuts down on the overhead for *get* operations, which matters most with small items (variable length metadata encoding is provided by Protocol Buffers [15] in the second system revision; see Section 6.6.2).

Figure 12 shows the overheads incurred using full variable length encoding with the second *temporal streams* wire protocol; even though the second wire protocol is less succinct and encodes redundant information (e.g., both the request data header and individual item metadata headers encodes item data lengths), it still requires 50% or fewer bytes compared to the fixed size metadata (84 bytes vs. 42 bytes in the largest case for this application). Additionally, a *get* request with typical parameters will require 52 bytes (50 byte header plus 2 byte header length) instead of 64 bytes – again, this small savings will matter most when each *get* request returns a small amount of data. Realistically, however, these savings are an ancillary benefit of using Protocol Buffers [15], as optimizing for small data items was not a major consideration; the decision to use Protocol Buffers was primarily based upon its superior flexibility, modern toolchain and more natural mapping into common programming languages.

**In Context:** To provide a frame of reference for our numbers, ASAP's [180] published end-to-end latency results for live queries are between 135-175 ms, which in practice are perfectly adequate for the application domain. In our *temporal streams*-based evaluation, the highest latency we have measured for historical queries is 18ms (+/-16ms), and live

[15]The item metadata size depends on the actual values of the item timestamp, duration and data size.

**Table 11:** System Data Overheads in bytes

| Data | Video | Face | Optical Flow | Sensor |
|---|---|---|---|---|
| Payload | 230,408 | 128 | 128 | 1024 |
| Response Header | 64 | 64 | 64 | 64 |
| Item Metadata | 20 | 20 | 20 | 20 |
| Total System Data | 84 | 84 | 84 | 84 |
| Total | 230,492 | 212 | 212 | 1108 |
| Overhead % | 0.036% | 39.62% | 39.62% | 7.58% |

**Table 12:** System Data Overheads in bytes (with variable length encoding)

| Data | Video | Face | Optical Flow | Sensor |
|---|---|---|---|---|
| Payload | 230,408 | 128 | 128 | 1024 |
| Response Header Len | 2 | 2 | 2 | 2 |
| Response Header | 13 | 13 | 13 | 13 |
| Data Header Len | 2 | 2 | 2 | 2 |
| Data Header | 4 | 3 | 3 | 3 |
| Item Metadata Len | 2 | 2 | 2 | 2 |
| Item Metadata[15] | 15-21 | 14-20 | 14-20 | 14-20 |
| Total System Data | 38-42 | 36-40 | 36-40 | 36-40 |
| Max Total | 230,450 | 168 | 168 | 1064 |
| Overhead % | 0.018% | 23.81% | 23.81% | 3.76% |

queries are even lower. Although the *temporal streams*-based implementation and functionality are not directly comparable to the ASAP system, the structure of our application components is derived from the ASAP design and both evaluations were run on the same hardware using the same OpenCV library primitives for analytics. This does show that the results are promising and potentially provide headroom for higher fidelity. These results also show that the *temporal streams* runtime adds minimal overhead to the baseline stream processing operations which are at the core of such stream analysis applications.

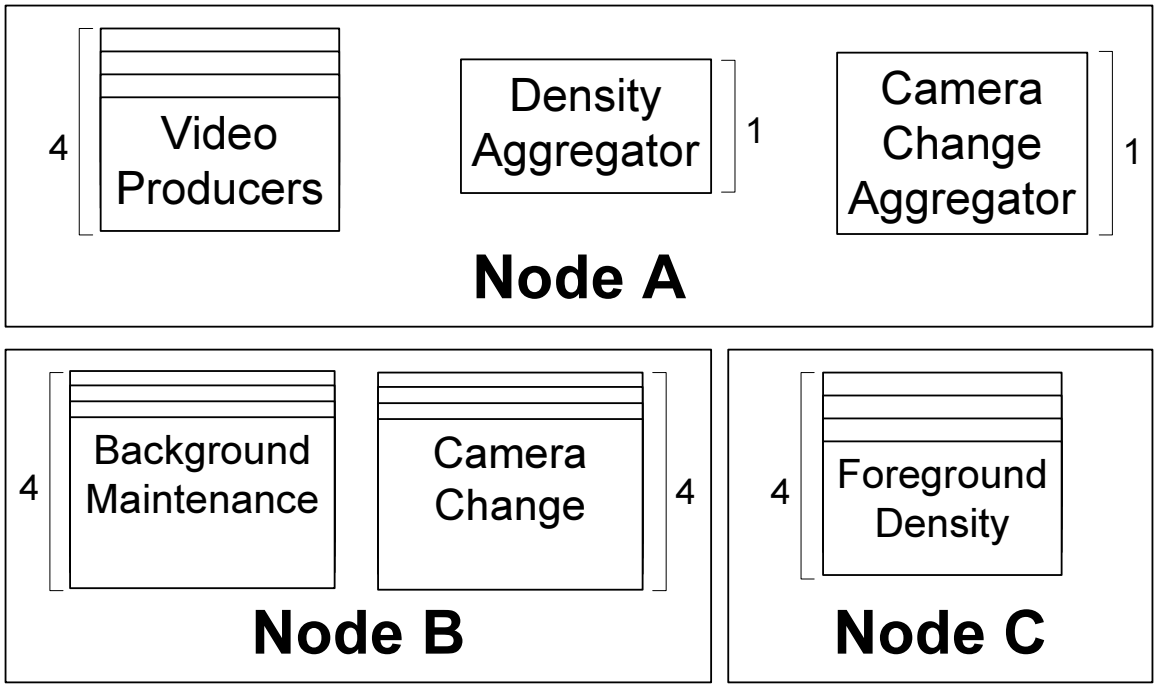
### 8.3.2 Traffic Monitoring

**Components Recap:** Our traffic monitoring scenario on *temporal streams* consists of five major parts: 1) video data producers, video feature detectors 2) background maintenance, 3) foreground separation, 4) camera change detection, and 5) feature aggregation components. Figure 50 depicts the dataflow between components. Video channels are hosted by the respective video data producer components. The video data items are 225KiB RGB video frames produced at 30fps. The video producers generate the data by decoding specially produced MJPEG compressed video files and playing them back on a loop (see "Video Data" below for a description of the input video).

The background maintenance component retrieves video frames one at a time from a video channel and maintains a running background model. It overlaps computation and communication by getting video data to process from a local replica of the video channel. The foreground separation and camera change detection components both consume data from the background maintenance component as well as the original raw video data (also using local replicas of the raw video data to overlap communication and computation). The foreground separation component uses the current background model and a video frame

to separate out the foreground content and then perform a traffic density estimate. The traffic density component's two input channels – background and video – are put into a channel group to virtually synchronize their contents. The camera change detection component gets raw video data normally and retrieves background data conditionally (based on changes detected). In our simulation, we force the camera change detection to retrieve background data every 90th frame, and for measurement purposes we limit the retrieval to a single frame. Finally, the feature aggregation components consume data from the traffic density (foreground separation) and camera change detection components to measure the processing latency. Each feature aggregator measures the latency for a small number of feature detector outputs, and each aggregator uses a dedicated thread per channel to gather more accurate information (using the concurrent feature aggregator pattern described in Figure 70).

**Topology:** In our setup, we host four video producers (and their corresponding video channels) per cluster node; all four producers are logically separate processes. Each video producer node will be decoding 4 MJPEG video streams to produce RGB video frames. It is also responsible for serving video content to twelve feature detectors (four of each type). The rest of the pieces run on other nodes in different groupings. The feature detectors run in groups of four per node and host their own output channels locally; the background maintenance and camera change detectors run on the same nodes (thus each node hosts eight components, four background maintenance and four camera change). The feature aggregators run on the same nodes as the corresponding video producers (i.e., an aggregator consuming foreground density information for cameras 5-8 runs on the same nodes as the producers for cameras 5-8) – this allows us to get full-pipeline latency measurements where the starting and ending time measurements are taken on the same physical node, lowering




This figure depicts the node assignment of components with four video channels. This setup would be replicated for every four video channels added to the system.

**Figure 72:** Traffic Monitoring – Topology

the potential clock skew. For our experiments, we use eight to twenty-four video streams total and each group of four video streams adds three cluster nodes. We use a total of 18 cluster nodes for the twenty-four stream configuration. Table 13 summarizes this setup. In order to run the experiment, we created a test harness that starts up the various components based on a declarative dataflow graph dependency representation. The corresponding file used for this set of experiments is presented in Figure 73; note the similarity to Table 13. For illustrative purposes, Figure 72 graphically depicts the component to node assignments for a four video channel subset of the experiment.

**Video Data:** The video data files used as input are custom generated for this experiment.

**Table 13:** Traffic experiment

| Component | Configuration | Total | +Nodes |
|---|---|---|---|
| Video Producers | 4 per node, one per stream | 24 | 6 |
| Background Maintenance | 4 per shared node | 24 | 6 |
| Camera Change | 4 per shared node | 24 | - |
| Foreground Density | 4 per dedicated node | 24 | 6 |
| Density Aggregators | 1 per video node, one per 4 streams | 6 | - |
| Camera Change Aggregators | 1 per video node, one per 4 streams | 6 | - |

The "+Nodes" column specifies the additional nodes required by each column. The addition of "Camera Change" components does not incur additional nodes because each group of four runs collocated on the node with a group of "Background Maintenance" processes. The aggregators run on the same nodes as their corresponding video producers.

```
# deps... component  | per machine, machine group id
VID VID              | 4, 0
VID BG               | 4, 1
BG VID CAMCH         | 4, 1
BG VID FG            | 4, 2
FG*4 AGG%1           | 1, 0
CAMCH*4 AGG%2        | 1, 0
```

Compare to Table 13.

**Figure 73:** Traffic experimental control file

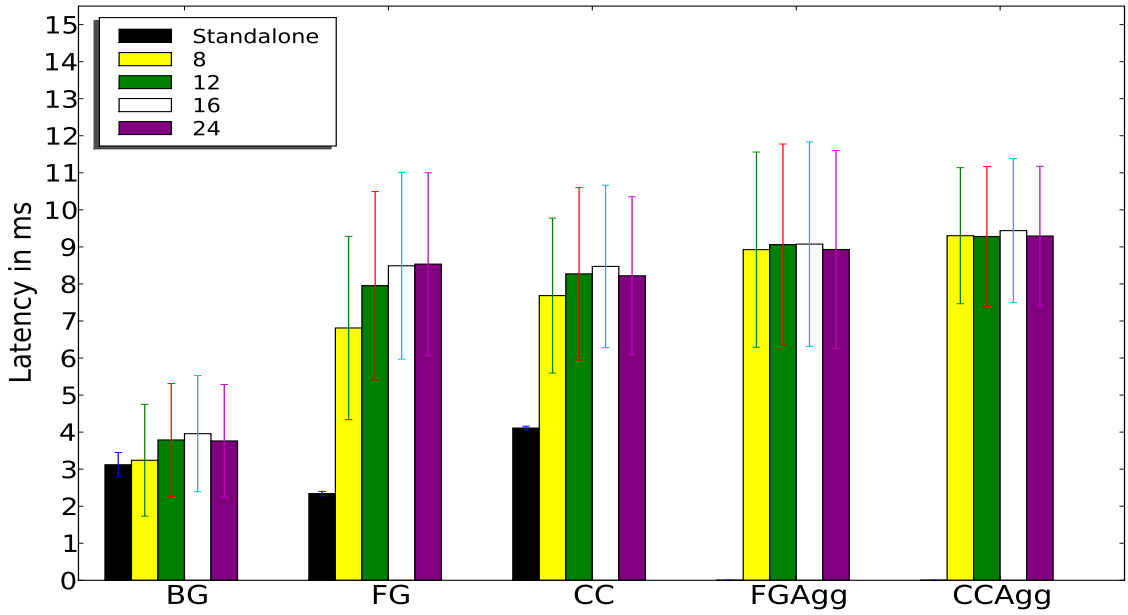



**Figure 74:** Traffic Monitoring – Component Latency in ms.

Bulk traffic camera still image output was gathered from the Georgia DOT's Navigator site[16] and used to generate static road background images. Using the road background images, we generated a variety of simulated traffic videos using moving foreground rectangles. Eight videos of varying levels of traffic density, four from daytime and four from nighttime, are used in our experiments. Each video stream is assigned one of the eight videos, round-robin. Each video frame is a 320x240 color JPEG image, and the frame rate is 30fps. The video producers decode this JPEG data to raw uncompressed video.

**Workload Characteristics:** We perform five runs of each experiment, each run lasting five minutes and 45 seconds and involving around 10,350 frames of video for each channel. We repeat the experiment scaling the number of video data channels from 8 to 24 and measure the processing latency of various components.

[16] `http://www.georgia-navigator.com/`

**Table 14:** Raw Data: Traffic Monitoring – Component Latency in ms.

| Config | BG | FG | CC | FGAgg | CCAgg |
|---|---|---|---|---|---|
| Standalone | 3.12 (0.33) | 2.34 (0.06) | 4.11 (0.05) | - | - |
| 8 | 3.24 (1.51) | 6.81 (2.47) | 7.69 (2.09) | 8.93 (2.64) | 9.30 (1.84) |
| 12 | 3.79 (1.53) | 7.95 (2.54) | 8.27 (2.33) | 9.06 (2.72) | 9.28 (1.89) |
| 16 | 3.96 (1.57) | 8.49 (2.52) | 8.47 (2.19) | 9.08 (2.76) | 9.44 (1.95) |
| 24 | 3.76 (1.52) | 8.54 (2.47) | 8.22 (2.13) | 8.93 (2.67) | 9.29 (1.88) |

**Component Latency Results:** Figure 74 shows the latency measured at different components in the application as the number of video streams is scaled from 8 to 24. We also present standalone measurements for the computation part of the three feature detectors run in isolation on unloaded (otherwise idle) nodes. The first column shows the processing latency measurements at the background maintenance component. The next two columns, "FG" and "CC" show the latency measurements at the foreground separation/traffic density components and the camera change components, respectively. The Agg columns show the measured latency at the foreground and camera change aggregators, respectively. In all cases, the latency is calculated using the timestamp of the original video frame (and averaged across all instances and all five runs).

First of all, when looking at the difference between standalone and application measures, consider the feature detector dataflow topology: the background maintenance component only includes one network hop (fetching video data from the producer), while the foreground and camera change components both consume data output from the background maintenance process. In particular, the foreground component shows the biggest increase in latency from the standalone case – this is expected as the foreground components are running on a separate node and consume channel-group synchronized data from both the background maintenance output and the original video output channels. This means that the critical path to consuming data includes two network transfers and one feature detector

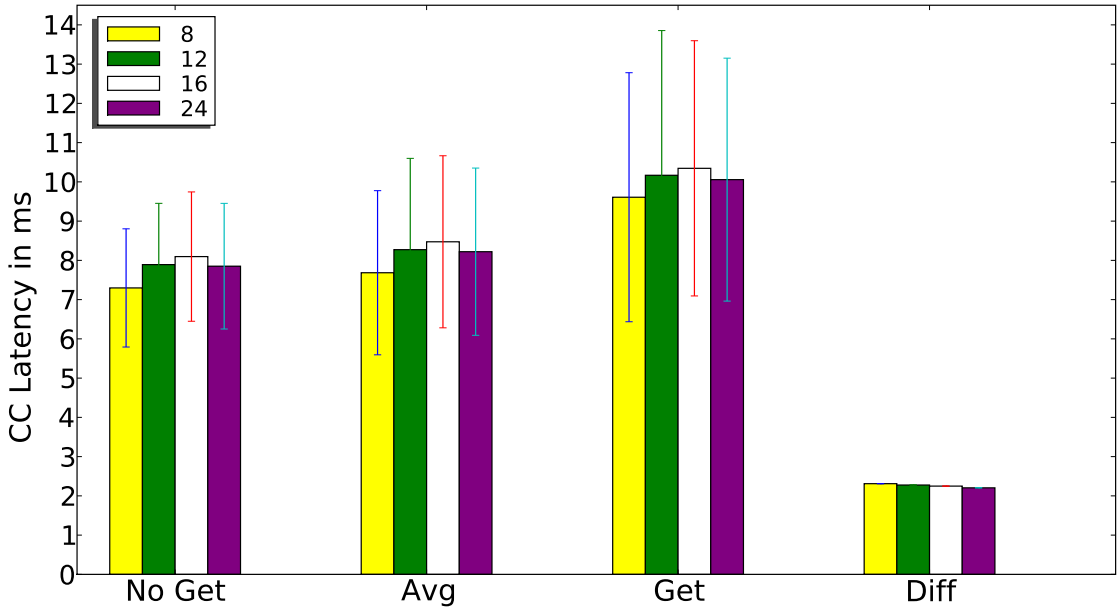
computation.[17]

On the other hand, the camera change detector primarily consumes data from the video producer, only getting background model data every 90th frame. We can break down our measurements of the latency into averages calculated with and without the background model *get* operations. These results are shown in Figure 75. The "Diff" column calculates the difference between the "Get" and "No Get" cases. We can see that the difference between these two cases is between 2.2 and 2.3ms. In addition, the "Get" measurements clearly exhibit higher variances incurred from an additional *get* operation. Although this is not a remote network operation, since the background maintenance and camera change processes run on the same node, it does potentially involve a blocking wait for background model processing. Note that the sampling method used to generate latency measurements during our experiments predictably over-weights the relative frequency of "Get" operations – although the background *get* operations occur only every 90 video frames, the application instrumentation only measures every 15th feature detector operation in order to cut down on the large volume of log data written.[18] This means that the "Get" operations will occur during 1/6th of all samples rather than 1/90th, but ultimately this difference is unimportant to our comparison.

In general, however, we can see that the latency variance between components increases directly with the number of network hops incurred (as expected). The key trend to note in Figure 74 is the fact that the aggregator latencies are essentially flat as the number of streams is scaled. This is expected as the *temporal streams* architecture does not contain

[17]The background maintenance must fetch data from the video producer and run the background calculation. Then the foreground separation component must fetch that background model output and a video frame from the original producer. Both of these are guaranteed to be remote network operations given the node assignments of the components (although there may be some partial overlap).

[18]Large volumes of logging output slow down and perturb the application's performance.




"Diff" shows the difference between "Get" and "No Get." The raw "Diff" data is as follows:

| 8 | 12 | 16 | 24 |
|---|---|---|---|
| 2.311 ms | 2.276 ms | 2.249 ms | 2.205 ms |

**Figure 75:** Traffic Monitoring – CC Latency in ms. (with and without gets)

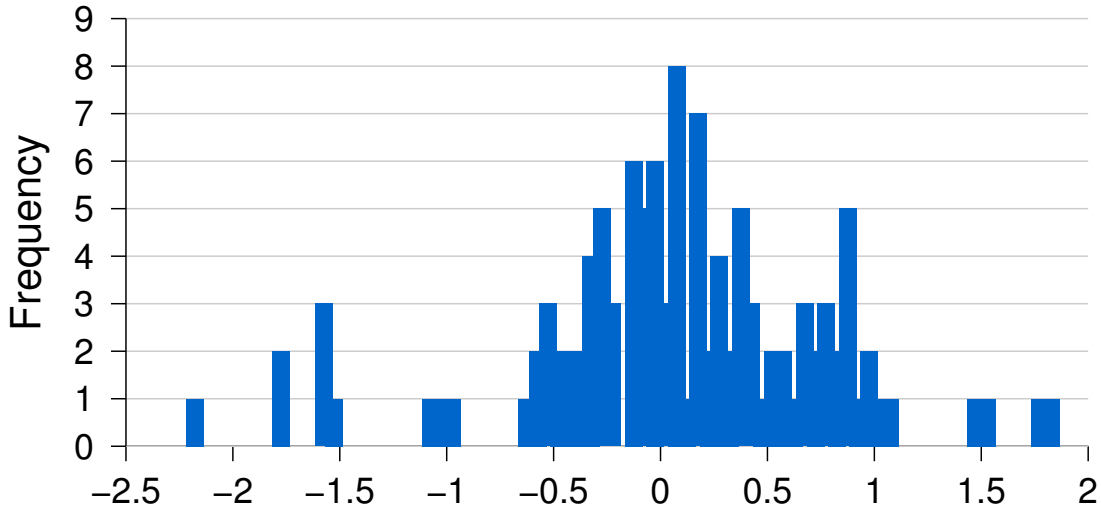



**Figure 76:** Measured NTP Offsets in seconds

any centralized bottlenecks, but these results demonstrate that fact quantitatively.[19] The aggregator results will also be the most accurate measurements since the starting timestamps (the video producer timestamps) and the ending timestamps (when the aggregator measures) are both measured on the same physical node, thus minimizing time skew due to slightly unsynchronized local clocks (see below for more on clock skew). The aggregators also use dedicated threads for each input channel to prevent one slow channel's blocking *get* operation from perturbing other channels' latency measurements (i.e., the concurrent feature aggregator pattern described in Figure 70).

**Time Skew:** The feature detector computation in the traffic monitoring experiment is an order of magnitude less expensive than the computation used in the airport surveillance experiment. The standalone feature detector latencies are on the order of 2-4ms rather than

[19]The only central points of contention imposed by the experiment are the network switches and the NFS mount where the output logs are sent (and the output log I/O is unlikely to be a significant issue at the levels of data involved in our experiments).

20-40ms in the airport surveillance scenario. This means that clock skew between nodes will be relatively more significant in our measurements. The aggregator measurements address this problem by taking starting and ending measurements on the same node. With the other measurements, however, we can clearly see the effect of clock skew. Figure 78 shows the 24 separate background maintenance latencies grouped by individual cluster nodes. We can see that there is a clear clustering effect where each node has a particular time skew but the measurements within a single node are quite close. This skew could also be caused by network-related issues, but we note that the average background maintenance component latency on some nodes (13-16 and 17-20) is actually below the standalone measured average of 3.12ms. It is unlikely that the addition of a network hop would significantly decrease the feature detection latency, so this points to clock skew effects. Compare the node-centric skews of Figure 78 with the 24 individual aggregator measurements for both feature detector types in Figure 79 and Figure 80. Despite the fact that these aggregator measurements go through a greater number of network hops, we see they are much closer.

In fact, independent sampling of the `ntpd`'s internally maintained skew measurements confirm skews on the order of ±2 ms between nodes. Figure 76 shows a histogram of skews measured on the experiment nodes at several different times – the `ntpd` estimated offsets from the primary time source are queried with `ntpd -p`. These offsets also vary significantly over time. To illustrate the effect of the varying offsets on our measurements, Figure 77 shows a limited subset of experimental configurations from Figure 74 compared with a second series of five runs of the eight video stream configuration. Note the significant difference in measured feature detector latencies between the first and second eight stream tests. Despite these variations, the aggregator latencies are still nearly identical. This highlights the value of taking aggregator measurements on the same nodes as the video

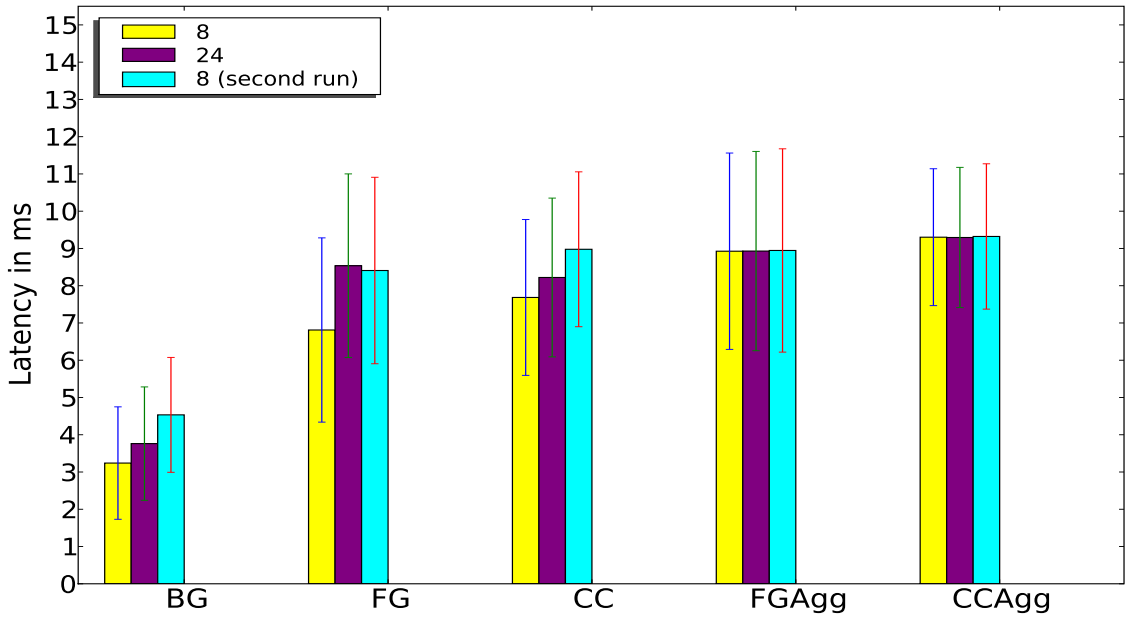



**Figure 77:** Traffic Monitoring – Component Latency in ms. (run variation example)

producers.

**In Context:** If we use the standalone computational cost of the feature detectors, we can do rough back-of-the-envelope calculations for the pipeline cost measured by the aggregators. The foreground aggregator should see the cost of the foreground computation (2.34 ms), plus some portion of the cost of the background computation (3.12 ms), plus the cost of transferring two video frames (the background model plus the raw video frame), plus some channel group synchronization delay, plus the network hop from the feature detector to the aggregator node. In practice, the cost of transferring two video frames will be overlapped with each other, or the transfer of the raw video frame may be overlapped with the background computation. Additionally, the background computation and foreground computation will be partially overlapped because the foreground computation for a frame at time $t$ will proceed as the background computation is occurring for a frame at time $t + 1$,

which is why we say it should only include a portion of the background computation. If we assume that any beneficial pipelining overlap is roughly equivalent to the channel group synchronization delay (canceling each other out), we can ballpark 2.34 + 3.12 = 5.46 ms as the computational cost. We estimate the average cost of the video frame transfer as 1.7 ms – double the ideal measurement of 0.85 ms on an unloaded node, adjusting for the computation and fact that many transfers are taking place. If we then estimate the cost of another network hop to the feature aggregator as ~0.85ms (lower than the video frame transfer because the data payload is much smaller), we get a total estimate of about 8 ms, which is about 1 ms smaller than our actual aggregator measurements.

The camera change computation is the simpler of the two – it should see the cost of the camera change computation (4.11 ms) plus the cost of transferring a video frame. It should also include the network hop from the feature detector to the aggregator node and some residual cost for the background model gets (~2.25 ms 1/6th of the time). If we add those figures up, we get $4.11+(2.25/6)+1.7+0.85 \simeq 7$ ms. Our actual measurements are between 9.3 – 9.4 ms, and it may seem odd that the estimated foreground latency is higher than the camera change latency when the actual aggregator measurements show the opposite trend (i.e., camera change measured slightly higher). One key unaccounted factor, however, is the node assignments of feature detector computation – the camera change computation is already more expensive than the foreground separation and it runs on a node that is much more contended (also running all background separation computation). The standalone measurements are executed on idle, un-contended nodes, so they are really lower bounds; even a modest 33% increase in average computational time due to the contention of running eight feature detectors and making many large network transfers would more than account for the difference.

These estimates are extremely rough, but they do give a very rough "ballpark" feel for the potential levels of overhead imposed by the *temporal streams* system implementation. These traffic monitoring experiments show that, even with feature detector computation on the order of 2-4 milliseconds per frame, the *temporal streams* library adds little overhead to our application-specific computation. The end-to-end, full-pipeline measured latencies are on the order of 9ms, and that includes several network hops and multiple potential steps of computation. Additionally, the fact that latencies remain flat as we scale the number of consumers shows the benefit of the system's decentralized design.

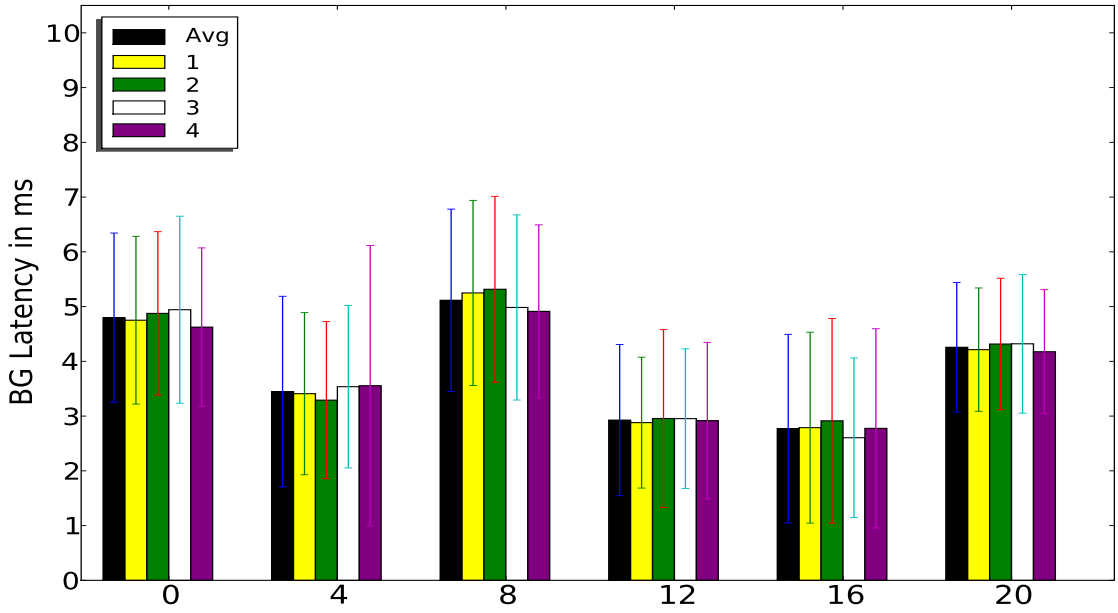



**Figure 78:** Traffic Monitoring – Background Latency in ms. for each node

Each node hosts four video background detection components. This graph depicts all 24 independent component instances, grouped according to node. The first bar in each group is the node average (averaging the four instances in each group). The groups are labeled with a number $x$ and each group contains components $x + i$ where $i$ is between 1 and 4. For example, the second cluster contains detectors 5-8 since it is labeled 4 (4 + 1, 4 + 2, ...).

**Table 15:** Raw Data: Traffic Monitoring – Background latency per node averages

| Group | Average in ms. |
|---|---|
| 1-4 | 4.799 |
| 5-8 | 3.448 |
| 9-12 | 5.116 |
| 13-16 | 2.927 |
| 17-20 | 2.771 |
| 21-24 | 4.257 |

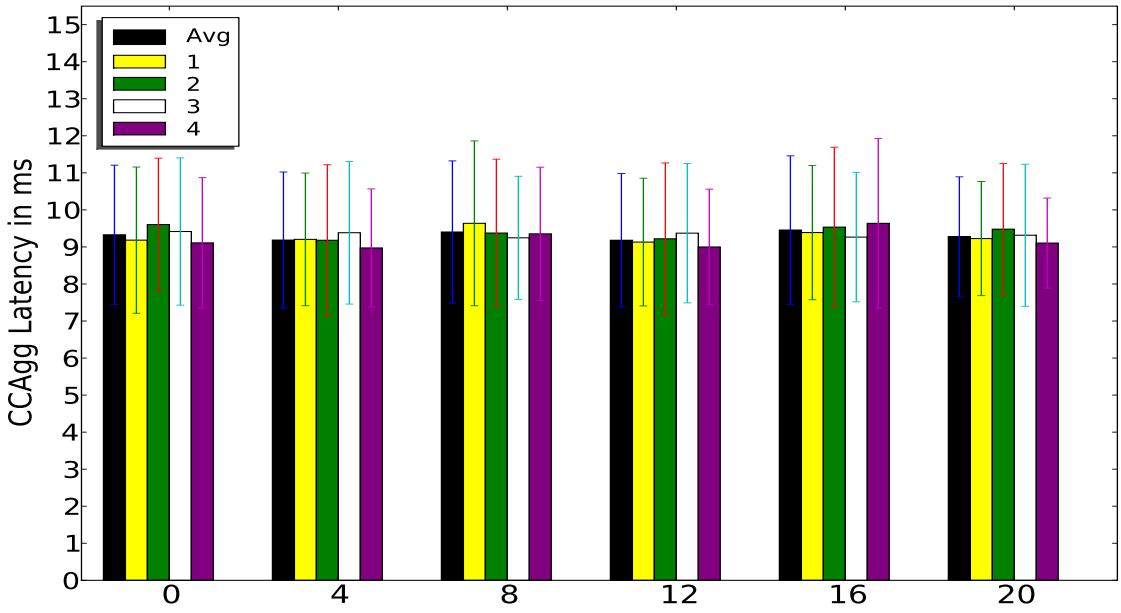



**Figure 79:** Traffic Monitoring – Camera Change Aggregation Latency in ms. for each node

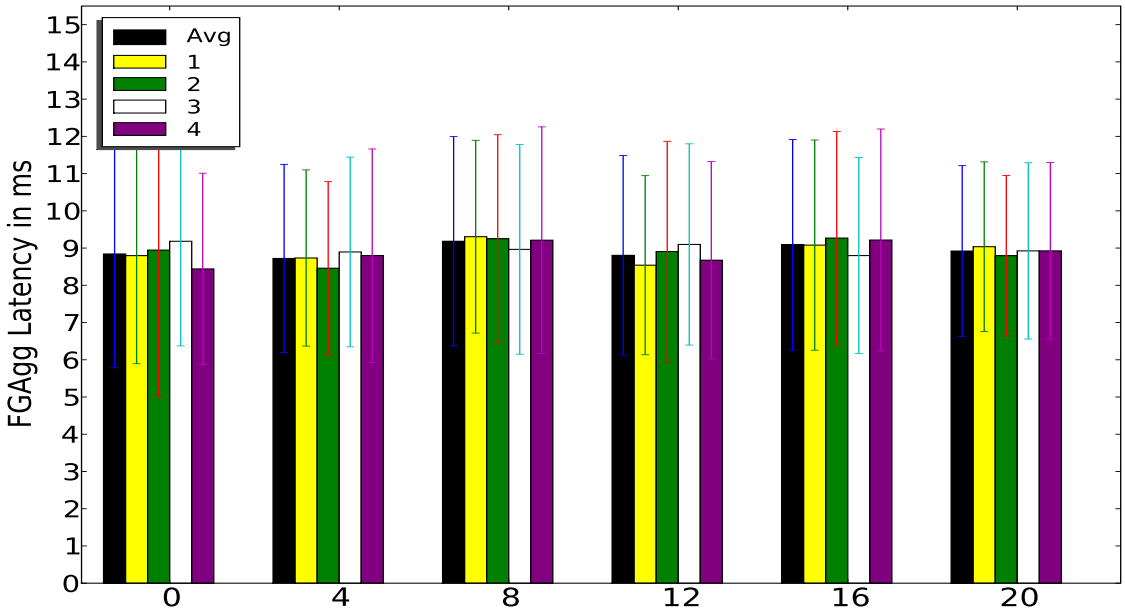



**Figure 80:** Traffic Monitoring – Foreground Aggregation Latency in ms. for each node

## *8.4 Discussion*

As described in Section 8.2.4, the three application scenarios highlight different properties of live stream analysis applications, spanning a range of potential use cases. Specific features of *temporal streams* directly support various stream analysis requirements. At the most basic level, the runtime provides streaming data transport and communication between distributed application components; data access uses temporal indexing specifically relevant to live streams. In the airport monitoring scenario, the application makes use of the programming model's stream data persistence and time-based historical data access mechanism. The airport application also uses the pickling handler functionality to transform live RGB video data suitable for analysis into a compressed (JPEG) representation suitable for storage.

In addition to the basic time-based data access, the traffic monitoring scenario uses a channel group to virtually synchronize a fundamental video stream with a feature stream created from the video stream. The camera change detector dynamically changes its number of inputs used in analysis. In steady state, with no changes, the detector uses only fundamental video stream input. When a potential change is suspected, the detector retrieves a set of background models spanning 10ms around the suspected change in order to do further analysis. The flexibility to choose arbitrary inputs and outputs dynamically is not universal in stream processing solutions; for example, systems based on synchronous dataflow models often quite restrictive and database-oriented solutions may not allow the expression of certain kinds of dynamic input changes or only do so in the context of user-provided operators which are opaque to the system's query more advanced optimization and query planning functionality (see Chapter 10 for more on related work).

In order to provide looser coupling and impose fewer restrictions on the stream computation, *temporal streams* does not provide application-wide scheduling of computation. In the traffic monitoring experiment, the video channels and related feature computation are statically partitioned onto cluster nodes. Although such a configuration may seem simplistic, it actually directly fits the topology of the GDOT's existing camera and data transport system. Although the traffic cameras are spread throughout the entire city and all feeds are sent back to the traffic monitoring center (TMC), there are also many local fiber concentration stations servicing a nearby geographic area. In these stations, a small number of nearby cameras' feeds are concentrated into a single data feed sent back to the central headquarters. The addition of a processing node to perform local feature detection on the small local set of cameras is a very natural extension of the existing infrastructure.

The port asset tracking scenario performs higher-level inferencing and anomaly detection using a semi-automated monitoring feedback strategy involving human operators. Even when operating with high-level events (versus lower-level feature streams), time information is critical; when we "close the loop" between higher-level event behavior and lower-level data to generate alerts or diagnostics for remediation, we use the *temporal streams*'s time-based retrieval to grab potentially large intervals of data spanning historical content. The system provides the ability to seamlessly express retrieval of live or stored stream data without making arbitrary distinctions. This allows the "Truck Leaving Port Late" rule action (Figure 63) to perform a binary search over a truck's journey through the port to find the initial source of a delay.

Although it provides key functionality to enable live stream analysis applications, the *temporal streams* runtime is designed to be relatively lightweight and suitable as a substrate for higher-level, more domain-specific middleware functionality. The runtime overhead

incurred by the core feature detection operations is negligible. In both of our experiments, the difference between the standalone computation time of the feature detectors and their computation time in our running application is on the order of less than a millisecond to a few milliseconds. This difference includes the time taken to transfer the data over a network, effects related to running on a busy system versus an idle system and scheduling jitter (and in some cases, waiting for synchronization or waiting for output from other feature detectors). In addition, the data overhead of *temporal streams* imposed by headers and metadata is relatively small (84 bytes) and can be further reduced by variable-length encoding.

**Robustness:** The process of running our experiments also highlighted the robustness of the runtime's decentralized design. When running our traffic monitoring experiments, we accidentally encountered a node without working Infiniband networking. Only internal *temporal streams* communication uses the Infiniband interfaces, while the test harness running the experiment uses the IPs associated with the regular external Ethernet interfaces to `ssh` in and spawn the appropriate processes. Consequently, the malfunctioning Infiniband interface was not detected until after the experiment had run. Communication to and from components running on the malfunctioning node was not possible. In our results we noticed that we had no measurements for one set of foreground density components and the upstream aggregators dependant on their outputs. Despite this failure, all of the other components in the system were unperturbed, including other feature detectors running on the same video streams. The traffic experiment's aggregators also use a dedicated thread per input channel (see Figure 70), so an unexpected delay in one channel's output will not block other measurements.

**Table 16:** Application Complexity

| Component | SLOC |
|---|---|
| Airport Surveillance (C) | 1050 |
| Traffic Monitoring (C) | 810 |
| Port Asset Tracking (DRL) | 75 |
| Port Asset Tracking (Java) | 250[20] |

**Complexity:** Table 16 shows complexity of each application measured in source lines of code (SLOC) as reported by SLOCCount [206]. These figures do not include the code related to OpenCV [14] computer vision library functionality. Since the airport surveillance application consists of seven components, and the traffic monitoring application consists of five components, we calculate an average of 150-160 SLOC of C per component. In a modern language like Java, with automatic memory management and a large standard library, the number of lines of could easily be halved. Ultimately, *temporal streams* does not eliminate all complexities in constructing live stream analysis applications, but it does provide a significant set of important baseline functionality. This enables relatively compact implementations of our motivating applications.

**Enhancements:** The experience of implementing these applications also highlights potential avenues of improvement in *temporal streams*. In particular, the traffic monitoring scenario's camera change detection (Figure 55) would benefit from time variables dealing with reverse time intervals. As described here, *temporal streams* provides the *next* and *newest-after* time variables, but does not define analogous, reverse-direction *previous* or *n-before* style variables. Currently, a *get* request can retrieve prior items only by subtracting concrete time amounts from known timestamps (e.g., in Figure 55, 10 milliseconds before

[20]This port asset tracking application implementation is more conceptual, since it is not a complete system; the SLOC figure provided here is counting the prototype code plus illustrative pseudocode stubs.

the point of interest). While this is adequate in some situations, it is also possible that the span of time to subtract to move back a certain number of items may not be known a priori. Enhancing the programming model with these time variables would not impose much on the implementation; the current system could accommodate such time variables already.

The port asset tracking scenario would benefit from the enhanced system stream registry discussed in Section 6.7. In that section, support for arbitrary application-level metadata associated with channels is described in the context of analysis priority and feedback. In the port asset tracking scenario, channels could be tagged with location information to allow components to query geographic relationships between sensors. Since our other scenarios also involve cyber-physical systems, they could all potentially use such facilities, depending on the nature of their higher-level analysis requirements.

At the level of implementation, our applications share common code patterns – for example, unary feature detectors which simply transform one stream to another are universal. Although *temporal streams* makes these components relatively small, we should further reduce the required boilerplate for very commonly-used patterns such as unary stream transformation by providing pre-built code "templates" in some form. C++ template-based metaprogramming can make the process relatively easy. As our experience implementing live stream analysis applications in different domains grows, an appropriate set of such commonly required patterns will become more evident.

## *8.5 Future Directions – Video as a Service*

One application scenario where we are currently applying *temporal streams* is the extension of live stream analysis to cloud computing services. *Video as a Service*[21] (VaaS) is a new

[21] *Video as a Service* is perhaps more aptly named ***Video Analytics*** *as a Service*.

project umbrella in the tradition of cloud computing *-as a service* offerings. The goal is providing video stream analytics as services in an environment that utilizes cloud computing resources. The nearly unlimited capacity of cloud infrastructure-level and platform-level service offerings can be leveraged for demand-based scaling. The VaaS computational platform provides common video analytics as metered, cloud-based services; it also uses the cloud as an optional scale-out strategy to augment local computational and storage resources.

Although the VaaS project is currently under active development, *temporal streams* provides a good building block for system implementation. Since *temporal streams* models stream data interactions only – acting as a glue between loosely-coupled communicating components – building a distributed system from disparate components in different administrative domains is not problematic. From a computational perspective, this flexibility enables service-compositional approaches to application construction. In a stream analysis application built using *Video as a Service*, user-provided analysis components will coexist with black box service-provided components for common video analysis. The VaaS-provided components are black boxes since they are metered services, so there are strong and natural communication boundaries between VaaS-components and user-provided pieces. *Temporal streams* provides a simple, uniform interface between stream components, and an HTTP-based interface to the system (described in Section 6.7.1) would fit well with current practice in constructing cloud services.

Other live stream analysis construction approaches are arguably less suited to this style of application, making more "closed world" assumptions or imposing higher coupling between application components. Compiler or language-oriented research typified by solutions such as StreamIt [193] or WaveScript [93] tend to require more global program

knowledge. WaveScript in particular uses whole-program, profile-driven optimization to decide how distribute operator computation among nodes a priori. Database-centric systems such as Borealis [27] tend to assume the entire system will function in a common runtime execution environment responsible for dynamically scheduling operators and orchestrating communication. See Chapter 10 for more related work context.

In addition to cloud computation, we would also like to utilize cloud storage in *Video as a Service*. As a simple and transparent first step towards cloud storage, the flexible persistence architecture of *temporal streams* allows a backend to use cloud as a large "backing store" for local disk storage. In fact, one of the first VaaS-umbrella projects is a *temporal streams* persistence backend using Amazon's S3 cloud storage, plus EC2 cloud computation and its EBS block storage.[22]

Ultimately, however, *Video as a Service* will point to potential improvements to *temporal streams* to better suit the capabilities exposed – and challenges imposed – by the cloud. Many of the concrete software architectural improvements mentioned in Section 6.7 are directly related to the demands of VaaS. In particular, the chained or advanced hierarchical storage is helpful to utilize cloud storage; the common channel-related metadata "whiteboard" allowing priority and feedback information to be published would also prove useful for *Video as a Service*-based applications. In addition, advanced access control is important when utilizing cloud computing because the system crosses administrative boundaries and is less controlled. Finally, the communications interface to channels may be enhanced to better support the black box metered cloud service paradigm. As mentioned earlier, "REST-like" [85] HTTP-based system interfaces are common in cloud services today. While *Video as a Service* requires some domain-specific computational scheduling

[22] See `http://aws.amazon.com/` for more information about Amazon's cloud services.

capabilities and other higher-level functionality, it will not all be added directly to *temporal streams*. Instead, *temporal streams* should provide hooks to effectively support such higher-level domain-specific functionality while remaining a flexible lower-level substrate for such systems.

# CHAPTER IX

# RETROSPECTIVE

In Chapters 7 and 8, we evaluate *temporal streams* in several ways: Chapter 7 focuses on the raw system performance, while Chapter 8 evaluates *temporal streams* in the context of several application scenarios. The application-based evaluations highlight how *temporal streams* provides needed functionality to upper layers while maintaining reasonable performance.

**Support for Upper Layers:** As mentioned in Section 2.3, *temporal streams* is designed to address a set of development "pain points" related to fundamental data issues in live stream analysis applications. Applying the concept of time to data access at a low system level elegantly solves a variety of common problems faced by applications in the domain. Fundamentally, *temporal streams* is a relatively thin stream *data* substrate for distributed applications; in previous sections, we have mentioned that *temporal streams* is designed to be suitable as a foundation for higher-level functionality.

Live stream analysis applications have a variety of concerns, and naturally *temporal streams* is not intended to address all of them. Higher layers – either the application itself or middleware layers built on top of *temporal streams* – will address additional application concerns. For example, the Port Asset Tracking application presented in Chapter 8 uses a business rules engine to reason about very high level events. Although the high-level needs of different target applications may vary significantly, *temporal streams* is sufficient to support upper layers' time-oriented stream data access needs.

In the Port Asset Tracking scenario, *temporal streams* is used directly by the application. In a system like ASAP [180], the highest-level application layer consists of queries written in a custom domain-specific language. These queries refer to time and stream data implicitly, so the use of *temporal streams* underneath is not directly exposed. Regardless, *temporal streams* is sufficient for the domain-specific language runtime implementation, which must compile queries into specific computation and related time-oriented data interactions.

Reasoning about high-level events is a very common concern in live stream analysis applications, particularly in cyber-physical systems. When low-level time-oriented streaming data is progressively refined into very high-level event data, there is a qualitative shift in the processing needs and uses of data. At the highest level, streams disappear to a large extent and the data is simply a sparse set of events; these events are associated with time but not necessarily processed in a stream-oriented fashion. At that level, non-stream-oriented event-reasoning systems such as Crest [30] are useful. However, as demonstrated in the Port Asset Tracking scenario, applications often need to "close the loop" and return to fundamental/lower-level stream data using time information associated with events, and *temporal streams* directly supports this need.

**Engineering and Use Considerations:** The *temporal streams* runtime (and programming model, to some extent) embodies certain assumptions about the system's use context. For example, *temporal streams* is designed to scale "out" as multiple streams are added, but its ability to scale "up" delivery of a single stream's data is not designed to the level required for something like Internet video delivery (where the number of consumers may be numbered in the hundreds of thousands or even millions). However, we believe that our facilities to scale up a single channel through replication are more than adequate for typical

live stream analysis.

As mentioned in Chapter 1, our primary focus is the subset of live stream analysis applications processing heavyweight fundamental input streams of relatively unstructured data where feature detection and analysis are the major computational requirements. Consider the standard structure of these applications as depicted in Figure 81. A typical application in this area has a moderate number of first-level feature detectors running on all input streams. These applications tend to be looking for high-level events of interest across a set of input streams, and the streams (like video) are not directly interpretable until relevant high-level information is extracted, so the system will usually perform some baseline processing on all input streams.[1] As these fundamental streams are transformed into progressively higher-level information, the amount of data is reduced. As higher-level information is generated, the outputs will often be more like sparse event streams of the type highlighted in the Port Asset Tracking scenario in Chapter 8. Higher-level analysis often combines multi-modal analysis or trend analysis; it may induce further selective processing on specific streams or request historical data from fundamental streams (also highlighted in the Port Asset Tracking scenario). Figure 81 shows this hierarchical processing as a one-directional processes with large amounts of data (at the bottom of the diagram) progressively transformed into more compact and higher-level information (at the top of the diagram). In reality there is a feedback between higher and lower levels, but the data volume trends still hold.

A consequence of this application structure is that the data demands are relatively uniform across streams, at least in the lower levels of feature extraction where data rates are

[1] Usually the application cannot know a priori which streams will be "interesting" or worthwhile to analyze until at least some first-level processing has occurred.

highest. In contrast, online user-generated video sites like YouTube handle streams with vastly different relative popularities. Online content consumed directly by users often follows power-law distributions [56, 209], which means that the relative popularity of streams may differ by many orders of magnitude. The current *temporal streams* architecture is not designed to handle data delivery at the scales required for those domains, but it is appropriate for live stream analysis applications.

The real world scenarios providing the basis for the applications presented in Chapter 8 involve 500-1500 video cameras. The results presented here show 8-24 streams to establish proof-of-concept. The Traffic Monitoring experiments show that the end-to-end feature detector processing latencies stay flat when scaling from 8 to 16 to 24 streams. I argue that the system's architecture should, by construction, allow scaling out the number of streams to these levels,[2] but obviously there is no concrete experimental proof.

This is a common caveat to academic systems research. Professor Ken Birman notes, "as a researcher interested in scale, I can't help but be disturbed by the tendency of [Program Committees] to demand types of experimentation that can only be undertaken by employees of the largest companies" [50]. If I could gain exclusive access to the cluster resources currently available to me, I could scale my experiments to perhaps 50-75 streams, but this is still an order of magnitude lower than the real world scenarios. While explicit experimental results are obviously a "gold standard," their potential infeasibility leads to an interesting question: how can one otherwise provide reasonably convincing evidence that a particular artifact can achieve suitable scale?

I conjecture that, while experimental evidence is key for evaluating a particular concrete system implementation, reasonably convincing evidence can be provided for a particular

[2] because channels are only related to each other either through system-wide metadata or channel groups

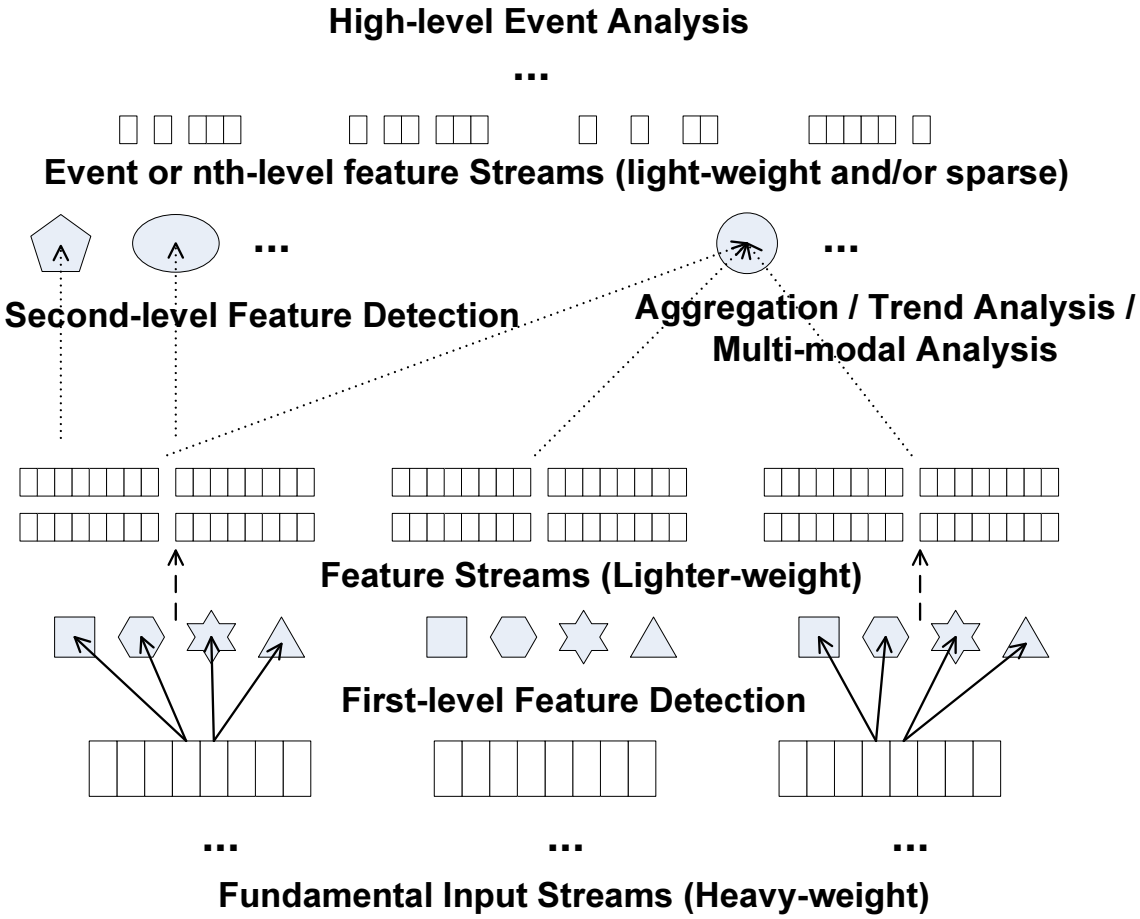



**Figure 81:** Structure of a Live Streaming Application

approach by evaluating the constraints imposed by general architectural properties. For example, this dissertation provides several successive refinements of the fundamental idea stated in my thesis. I move from 1) the abstract idea of wall clock time indexing in live streams to 2) a specific programming model using wall clock time to 3) a specific runtime architecture to 4) a concrete implementation of the architecture. At each level of refinement, certain constraints are imposed on *any* potential implementation. The first level of detail – the abstract idea – imposes certain overheads on individual streams, because time-indexing must be provided: this could potentially limit the ability of a single stream to scale "up" in aggregate data rate. However, it does not make any particular impositions on scaling "out" by adding multiple streams, as there are no specific constraints on how multiple streams affect each other. At the second level of refinement – the programming model – multiple streams are related loosely by global naming metadata, but again few constraints are imposed. Finally, at the level of an architecture, the programming model could be realized in an extremely non-scalable way by centralizing all resources on a single node or in an extremely scalable way by making all channels independent.

I have chosen to design a highly decentralized architecture for *temporal streams*. In general, the only architecturally-imposed interdependencies between channels are 1) channel groups and 2) global metadata. In Section 6.1, I note several limitations of channel groups and their use context: they are not intended to scale to a large number of channels or widely distributed channels, but such synchronization needs are already constrained. In our problem context, the message propagation delay between distributed nodes already limits the achievable synchronization accuracy. In addition, while channel groups may not scale to a large degree by themselves, this limitation only applies *within* a channel group. In other words, the presence of channel groups does not limit the ability of the greater system

to scale.

The performance of global *temporal streams* metadata really constitutes the major architecturally-imposed scalability limit when scaling "out" to many channels. I argue that it will not be an impediment to scaling to the levels needed by the application scenarios; as the system is designed currently, global metadata is queried infrequently and updated even more rarely. In many ways, the metadata facilities in *temporal streams* are similar to the Domain Name System (DNS): queries are relatively rare and updates are infrequent, hierarchical caching at clients is used to lessen the load on authoritative, top-level sources and the system uses eventual consistency. If the system's view of channels was more significantly more dynamic (for example, supporting dynamic publish/subscribe predicates as described in Section 6.7.2), a different approach might be necessary.

As the system is presented here, I believe one can reasonably accept that the abstract approach described in my thesis does not impose fundamental limitations that would prevent scaling an implementation "out" to many streams. While a particular implementation's scaling properties may vary, I believe it is evident that the programming model and architecture do not place significant constraints which would fundamentally prevent such scaling.

# CHAPTER X

# RELATED WORK

This chapter is a survey of existing literature relevant to our *temporal streams* programming model and runtime architecture. Since our work addresses a variety of different concerns, including high-level distributed programming models, networking issues, and storage of streaming data, this chapter covers a relatively broad and diverse set of efforts, spanning areas like databases, distributed systems/networking, programming languages and compilers. In this chapter, we attempt to be thorough but by no means exhaustive. Note that 1) some prior work is only related to a particular piece of our system architecture and 2) the works are roughly categorized – certain systems are thematically suited to several different categorizations. See Section 10.7 for a broad discussion of the thematic relationship of different classes of work.

## *10.1 Stream Databases & Stream Processing Engines*

Much work in alleviating higher-level concerns for live stream analysis applications comes in the form of *stream data management systems* (SDMS) (also referred to as *stream databases*) or *stream processing engines* / *data stream processing systems* (DSPS). These systems manage execution (and communication) of stream analysis functionality and often use *continuous queries* in declarative query languages. Many are evolved out of traditional relational databases. There are a variety of relevant research systems like domain-specific

Gigascope [68] and Hyperion[1] [75] (for network monitoring) as well as more general-purpose systems like TelegraphCQ [63], Synergy [169] and Borealis [27] (and its predecessor Aurora [43]), and Stanford's STREAM Data Manager [37]. IBM's Stream Processing Core [34] (part of System S) is another research system for *stream mining* (data mining on streaming data); it uses a publish/subscribe-inspired stream model and provides lower-level core runtime infrastructure for stream processing engines and applications. SPADE (Stream Processing Application Declarative Engine) [88] is a declarative stream processing engine that runs on the Stream Processing Core. NexusDS [65] is a recent DSPS designed to address various limitations in many of the aforementioned systems, such as support for unstructured streams, heterogeneity in processing nodes and dynamic addition of operators at runtime.

General-purpose commercial systems include StreamBase [17], Coral8 [6] and Truviso's TruCQ [18]. IBM's Middleware for Large-Scale Surveillance (MILS) [12] uses a regular DB2 database and targets surveillance and related applications specifically. In these systems, all streaming data analysis runs as part of the database or some sort of stream query engine. Generally, application-specific data analysis code is written as a stream *operator* in some extension API if user-defined operators are possible.[2]

Another series of related systems exist in a sub-community built around the area of *Complex Event Processing* (CEP) [131] or *Event Stream Processing* (ESP).[3] Systems for

[1] See Section 10.6 for more on Hyperion. Although it is heavily motivated by stream database approaches, the primary contribution in the cited publication is at a lower level – a high performance filesystem for stream storage.

[2] Otherwise the user is limited to the built-in query language features.

[3] The terminology is not well standardized, but CEP applied to event streams is often called *Event Stream Processing* (ESP). Some sources differentiate between CEP as a set of techniques for event analysis and ESP as application of CEP to streams [130], while others use the terms more interchangeably.

CEP are often similar to stream processing engines (or stream databases) and the distinction is not clear-cut; the concerns and applications often overlap and many differences are in the originating research community and terminology. CEP/ESP scenarios often involve a massive number of small data items (events) which are aggregated and correlated with pattern matching languages or rule engines – these are similar to continuous query languages, although many are explicitly limited in expressiveness for performance considerations [9].

Some systems explicitly targeted towards CEP include the research efforts SASE [97], Cayuga [74], and Microsoft Research's CEDR [46], as well as the open-source, but commercially supported Esper [23], the freely-available, closed-source Stream*Cruncher* [110] engine and the commercial ruleCore CEP Server [16]. Both StreamBase [17] and Coral8 [6] are marketed as CEP systems, although StreamBase's lineage is in Borealis [27]/Aurora [43] and Coral8's lineage is in STREAM [37]. Owens [159] surveys a large set of stream processing systems, including CEP systems. Dekkers also surveys [73] several competing CEP systems.

While these systems are impressive, they represent a different architectural approach than *temporal streams*. These systems provide centrally managed and controlled execution (often with high level query languages), while our system is a glue for loosely coupled systems of independent communicating components with no centralized control where distributed actors communicate implicitly via channels. *Temporal streams* does not impose any particular computational model on applications; it only models stream data interactions. Our system is also targeted at scenarios involving significant signal feature detection/analysis on streams such as audio and video, in contrast to SQL-like declarative query languages often more suited to domains with highly structured data like network monitoring or stock trading. The authors of the WaveScript language [93]/XStream engine [94]

note that traditional stream database approaches are not well suited to signal analysis applications (audio, for example) and provide an augmented model and stream management system for isochronous signal processing. The Linear Road benchmark [39], the only standard benchmark for stream databases/stream processing engines, uses highly structured data for analysis and does not include signal analysis or feature detection from data sources like video.

Perhaps more importantly, unlike traditional databases with SQL, there is no widely agreed-upon, "industry standard" temporal query language: CQL [38], TSQL2 [185], StreamSQL [22] (used in StreamBase [17]), CCL [5] (used in Coral8 [6]), and various other extended SQLs such as GSQL [68] and non-SQL based languages such as TQuel [184] (based on Quel, the original Ingres query language) have all been proposed over the span of the past twenty years. ATLaS [204] supports streaming and regular relational queries using an extended, Turing-complete SQL dialect. Even the baseline semantics of continuous queries regarding time windowing and blocking/non-blocking operators are still debated and vary from solution to solution; additionally, the suitability of various query semantics is potentially domain-specific. For example, the Gigascope [68] authors found that continuous queries with sliding windows – the most popular streaming database paradigm – are "inappropriate for network data analysis," so Gigscope instead uses a pure stream model. The recently published "Towards a Streaming SQL Standard" [109] proposes a unified execution model for time-based and tuple-based stream query languages.

Our approach does not impose a particular computational model on stream analysis applications; we only model stream *data* interactions, supporting arbitrary communication/data dependencies between components at the expense of being less declarative. In the end, we believe this tradeoff is acceptable given the additional flexibility of general

distributed applications (e.g., components can be developed independently/in different languages, hold internal state, utilize external resources, etc.). It is also easier to integrate into existing distributed systems, as components have explicit communication boundaries and are not executed in the context of a constrained execution environment. This, in turn, facilities service-composition approaches to application development. Our approach also provides a substrate for higher-level domain-specific solutions which raise the level of abstraction for a set of applications.

Ultimately, our approach represents another point in the design space balancing tradeoffs between flexibility/generality as well as performance and the level of abstraction. Our choices are similar to Distributed Data Structures [95] and BerkeleyDB [155], where some higher-level and heavyweight features of a full DBMS are traded off for a simpler, more procedural programmatic interface. In some ways our approach is similar to Boxwood [132], which provides distributed, managed data structures as a fundamental storage abstraction in lieu of more traditional approaches; in our case, the stream abstraction also serves as the storage interface. Distributed Data Structures [95] and Boxwood [132] also fit as distributed programming solutions (see Section 10.4).

## *10.2 Data Parallel & Stream Languages*

Two related classes of language and compiler-oriented research have adopted terminology involving "streams" and "streaming." I will call one category *stream programming* and the other category *continuous stream languages*. Works in the former category have evolved out of *data parallel languages* and those in the latter are typically derived from the *synchronous dataflow* model of computation. Both classes of solutions also share common traits and goals – one major goal is providing a means to express parallelism explicitly to

enhance optimization. Despite the overloaded common terminology, the groups evolved from somewhat different starting points. See Section 10.7 for a broad, high-level comparison.

### 10.2.1 Stream Programming

Presently, *stream programming* is a popular research area, with much current work on parallel programming models to effectively utilize GPUs for general purpose computation, the Cell processor, DSPs, custom FPGAs, many-core processors, and Polymorphous Computing Architectures [70]. In these models, stream processing is effectively an extension of SIMD (or MIMD), a mechanism for expressing and exploiting data parallelism. There is a proliferation of recent data parallel programming models and languages, including Brook [57], Peakstream [21], Accelerator [190], Ct [92, 91], CUDA [150]. These models are often similar to earlier languages such as APL [108] and NESL [53]. Recent evolutionary works such as Sequoia [83], OpenCL and Streamwave [96] go much further, extending the generality of computation and adding runtime support related to many general-purpose parallel programming systems. The main differences between *temporal streams* and these stream programming models are 1) the assumed level of available data parallelism, 2) the level at which the computation is modeled, and 3) the use of time.

Although there is significant variety in the specifics of the various stream programming models/languages, their basic model of streams are similar – streams are generally represented as data parallel arrays which are inputs to kernels of computation applied elementwise.[4] These models are typically applied to large-scale data processing tasks where the

[4]Some of these models (e.g., Ct or NESL) also support nesting in addition to "flat" data parallelism.

primary concern is scaling out to effectively utilize large amounts of available parallel processing power, such as in the massive number of ALUs available in GPUs – these approaches assume large amounts of data available for bulk processing. In addition, these systems generally have no notion of time and operate on bounded sets of data. Live streams are different, because data is processed as it is produced, so data parallelism between items is limited by the production rate and latency expectations (in waiting for results to be processed). In this manner these models are different, but our temporal stream model is also at a higher-level: the domain of the actual stream analysis code is entirely the application's. *Temporal streams* models the data production characteristics, while the actual processing of the data in a particular stream (or streams) could reasonably utilize data-parallel computation techniques. Since these approaches are at different levels of abstraction, they can be mutually complementary – many applications using temporal streams will also need to incorporate such low-level processing optimizations, and our higher-level abstractions are designed with that possibility in mind. By not defining an execution model for the individual stream analysis code itself, our temporal stream model can easily support exotic hardware and heterogeneous systems, provided the scheduling/resource management component is adequately aware of such issues.

SPADE [88] has nascent support for integrating stream processing techniques in the context of live stream analysis applications. It supports these techniques in the form of list data types, which are very similar to data parallel arrays of previous systems. They provide data parallelism for dispatch on vector extensions on general purpose processors, as well as specific support for the Cell processor. The authors of SPADE also compare the list data type approach for live stream analysis applications to WaveScript's [94] primitives for isochronous signal processing.

### 10.2.2 Continuous Stream Languages

StreamIt [193] is a stream language and compiler frequently compared to the aforementioned stream programming models; its goals are similar but StreamIt's view of streams is somewhat different. Rather than viewing streams as parallel arrays of data items processed by idempotent compute kernels, streams are an unbounded series of data items arriving in a particular order and at a known rate – this is the *synchronous dataflow* model [121]. This model facilitates static analysis and is exploited for automatic extraction of parallelism in moderate sized, structured tasks (such as sorts, media decoders, FFTs, etc.) decomposed into smaller filter hierarchies. The synchronous dataflow model of streams is closer to our model than the stream programming model, because streams are continuous and time-oriented; however, our model is asynchronous like most streaming database work. Various other systems similar to StreamIt bridge the gap between synchronous dataflow stream programming and asynchronous stream data management. Flask [133] is a domain-specific dataflow programming language for sensor networks. The WaveScript language [93, 94] is also similar to StreamIt in many aspects but provides elements of asynchrony like streaming database work.

Spidle [66] takes a dataflow-based, domain-specific language approach to supporting non-distributed streaming applications. By constructing streaming applications as networks of filter hierarchies in a declarative language, Spidle allows succinct specification of such applications and enables domain-specific static analysis, including verification. Nizza [189] provides a framework for the construction of real-time, non-distributed streaming multimedia applications. Nizza's approach is based on using dataflow-oriented program representations to exploit both task and data parallelism on SMP systems. TStreams [114] is a proposed model of parallel computation used as a language/compilation target – it

defies clean categorization. Mandviwala [134] describes TStreams as follows:

> The TStreams [38] parallel programming model was developed by combining ideas from Dataflow [4], Tuple Spaces [21] and Streaming computations [53, 60] to enable a model where parallelism and data-dependencies could be cleanly expressed separately from the distribution and scheduling policies of concurrent tasks. In TStreams, the application programmer expresses all the available potential parallelism by describing fine-grain computations and data communication specification between them via an application task-graph.

## *10.3 Processing Streams / FIFOs*

The general concept of processing potentially unbounded streamed data as it is made available is fundamental – for example, the Unix pipe[5] [172] is a nearly ubiquitous streaming data flow abstraction, as are lazily-evaluated infinite lists in functional programming languages [28, 113, 194]. Dataflow-driven languages such as Lucid [41], SISAL [139] and Id [40] use similar idioms as a core programming paradigm.[6] *Functional reactive programming* [152] integrates a form of dataflow programming dealing with time-sensitive asynchronous events into functional programming. Shivers and Might [181] explore techniques for optimizing pipelines of *online transducers*. An *online transducer* performs item-at-a-time computation on a perpetual data streams. This paradigm is related to Sawzall [163], a language made for writing mapping and aggregation functions in MapReduce [71] (see Section 10.5). McIlroy [140], inventor of the Unix pipe, shows how to evaluate power

[5] or any number of equivalent IPC abstractions for other operating systems

[6] Thies [192] explores the relationship between various classes of dataflow languages and models of computation in depth, and their connection to the previously mentioned classes of stream languages (Section 10.2).

series in Pike's Newsqueak language [164] in a similar manner (composing sequences of transducers applied to infinite streams).

Although they were introduced primarily to facilitate network protocol and service innovation, *active networks* [191] process data as it flows through the network en-route to its destination. There is an entire area of research in *online algorithms* or *streaming algorithms* in theoretical computer science [146]. Hundreds of other abstractions in many diverse areas also model streams as a sequence of bytes or messages and model computation as functions applied to perpetual input streams.

## *10.4 Distributed Programming*

The workload of live stream analysis applications is unique and lends itself well to distributed programming, because stream processing has natural and explicit communication boundaries. In addition, live stream analysis applications are also amenable to distribution because all shared state can be explicitly pushed into streams, which simplifies the analysis code into stateless, functional operations.

The development of more expressive and convenient programming models to ease the burden of distributed programming is a very common goal and there is an extremely large body of prior work in this area. The communications aspects of temporal stream abstractions are related to various distributed programming systems and programming models, such as message-passing systems, distributed shared memory, RPC/RMI, group communication, tuple spaces, or publish/subscribe systems. Nonetheless, no existing system directly provides a programming model tailored for the natural expression of temporal idioms common in the development of live stream analysis applications. At the lowest level, network

communication via sockets or byte streams provides high performance and excellent flexibility but is inconvenient and complicated. Much work has gone into layering more convenient and structured abstractions on top of underlying unstructured transports. Remote Procedure Call [186, 51] and Remote Method Invocation [11] are now nearly ubiquitous mechanisms to provide more structured network programming. Although very expressive and widely applicable, RPC/RMI are very general and typical implementations are unsuited for continuous bulk data transfers; the programming model of RPC makes it unnatural or impractical to exploit multicast for many kinds of interactions. Additionally, time-based manipulation of streaming data must be layered on top of a basic RPC system.

**Low-overhead Message Passing:** In many domains involving high-performance computing, message-passing systems are more common than RPC/RMI. Active Messages [202] and related systems like Optimistic Active Messages [203] and Fast Messages [160] are high performance point-to-point messaging layers, but they are low level and often more suitable as a substrate for implementing higher-level abstractions. Systems like PVM [78] and MPI [87] provide more facilities for point-to-point messaging and various collective communication operations. Although significantly more convenient than raw transport-level operations and very general, message passing systems like MPI and PVM are still fairly low-level; additionally, such systems have traditionally been narrowly targeted towards relatively static cluster-computing environments and may not handle failure or dynamism in a manner appropriate for more widely distributed environments. Various efforts have attempted to address related shortcomings: MPI-2 [141] addresses the issue of static participants by expanding the process model to allow runtime dynamism. FT-MPI [82] stands for "Fault Tolerant MPI" and attempts to address MPI's shortcomings with regard to failure tolerance.

**Group & Configurable Communication:** Configurable communication systems like Isis [49] and Horus [195] are often targeted for group/collective communication, but are more appropriate for applications requiring heavyweight features such as group membership agreement or causal message ordering. detection of group partitions and atomic message delivery (via virtual synchrony). CCL [42] also provides a number of powerful primitives for group communication. For our target class of applications, direct point-to-point message passing is inappropriate because it would require producers to be informed of the current consumers of a particular stream. In the target class of applications, group communication is more appropriate than point-to-point messaging, but per-stream group broadcasts would still involve much redundant messaging because each item would be broadcast to each stream consumer, even those that may not need it. Additionally, many group communication systems are not designed to support a large number of groups or groups with quickly varying membership. Recent related projects like QSM [158] attempt to address some of these limitations. Finally, the recognition of time as a first-class entity would still have to be layered on top of a group-based communication system.

**Publish/Subscribe & Message Queuing:** Publish/subscribe is a messaging paradigm similar to group communication – in fact, one can view group communication as a specific simple form of publish/subscribe, although in practice systems utilizing the two approaches often target quite different domains and therefore have different performance characteristics and design tradeoffs. Producers publish messages and all interested consumers receive messages; consumer interest can be determined in a variety of ways, but content-based classification or subject-based tagging are common. The Information Bus [154] (TIB/Rendezvous), Siena [62] and Gryphon [45] are notable examples of pub/sub systems. The Cayuga [74] streaming event processing system augments a publish/subscribe model with

a database-inspired, expressive query language and IBM's Stream Processing Core [34] also uses a publish/subscribe-based stream system. Message-queuing/brokering systems are also related to publish/subscribe and group communication systems and many also present basic forms of distributed data structures. Often both point-to-point direct messaging and publish/subscribe style distribution coexist in "message-oriented middleware" [44] solutions. Many such systems exist, but Sun's Java Message Service (JMS) [99] and AMQP [197] (Advanced Message Queuing Protocol) are two recent standards with a significant number of independent implementations and vendor support. JMS is simply an interface specification for Java messaging components enabling pluggable implementations but not necessarily interoperability, while the newer AMQP includes complete protocol-level specifications. Such systems are typically used for asynchronous event notification rather than bulk data transfer, and designed for business-oriented applications rather than high-performance computing scenarios.

**Distributed Shared Memory:** Distributed programming models with implicit communication, such as Distributed Shared Memory (DSM) and tuple-space environments like Linda [61], are also popular. DSM is not appropriate for our target class of applications due to the nature of data and the granularity of sharing. The rate of data turnover is high, but items are read-only once produced and any given consumer may simply need the latest item. DSM systems require relatively infrequent page faults to achieve good performance. Also, when the item size is greater than a single page (for instance, uncompressed frames of video), the system will generally require several remote operations to retrieve a single item. Making the page size large is also undesirable because it can cause false sharing/unnecessary data transfer of unneeded old data items or internal fragmentation for streams with smaller items. More fundamentally, stream-based data processing does not map naturally

onto shared memory models.

**Tuple Spaces & Language-based distribution:** Although they are not traditionally used for applications with high-volume communication requirements, tuple-space programming models like Linda [61] can provide a fairly natural mental model for stream-based processing if each stream is represented by a tuple space and a timestamp is used as the tag for each item. In order to provide automatic storage management, the runtime would need to keep track of a window of currency and reclaim items with timestamps older than the minimum bound. To the best of our knowledge, no existing tuple space implementation provides all of these features with suitable performance for real-time streaming media. TStreams [114] is a proposed model of parallel computation general enough to subsume the previous description of stream-based programming in tuple-based systems, but it is a language/compilation target to expose parallelism rather than a runtime environment. Some programming languages, such as Erlang [199] and Oz [100], provide communication channels or distributed message passing as core language primitives.

**Stream Delivery:** Many systems exist for transport of streaming data for viewing: Yima [179] and related systems are concerned with scalable media delivery to many clients. Several systems, such as TOAST [86], MAESTRO [105] and CORBA *Audio/Video Streaming Service* [118] augment CORBA [153] to provide transport of streaming media and concentrate on efficient transmission. Systems such as these often use lower-level transport protocols also designed specifically for real-time streaming media such as RTP [178], RTMP, RTMFP [25], and SCTP [157].[7] Various peer-to-peer systems also target streaming media in different ways: CoolStreaming/DONet [210, 124] uses an overlay network to distributed

[7]Here *transport* is used in a general sense: only SCTP is a proper transport protocol. The other listed media protocols are layered on TCP or UDP.

streaming video data, while Skype [47] provides VoIP service. RTMFP [25] also has peer-to-peer aspects. Liu et al. [129] survey peer-to-peer video distribution systems. Such data distribution techniques and protocols can be applicable in the efficient implementation of *temporal streams*.

**Information Flow:** Infopipes [52] are high-level communication abstractions for building distributed streaming multimedia systems. Infopipes focus on dataflow and communication in such systems, making the information-flow relationships explicit through compositional first class communication endpoint components which are significantly higher-level than transport primitives. By explicitly representing the program's dataflow with a vocabulary of higher-level communication design patterns (i.e., concepts such as "tee" connections, buffers, pumps and filters), the system can provide automatic management of QoS-related issues, scheduling of certain types of computation and data movement, as well as verification of certain application-level properties. DirectFlow [128] extends the concept of InfoPipes to a full domain-specific language for describing dataflow relationships. STAGES [207] also uses a domain-specific language approach to specifying dataflow graphs in interactive multimedia applications running on high-performance computing resources. In particular, STAGES uses an aspect-oriented programming approach to allow applications to separate core algorithmic components from cross-cutting concerns like connection topologies, parallelization strategies and the mapping of threads and data layout to available resources. These systems straddle the boundary between compiler and language-oriented stream systems and distributed programming solutions.

**Space-Time Memory:** Stampede [167] and D-Stampede [31] are the most closely related programming systems and are Stampede$^{RT}$'s [104]/*temporal streams*'s direct predecessors,

but there are significant differences in the programming models, especially in the handling of time and garbage collection. Stampede's "Space-Time Memory" programming model also involved distributed data structures, but applications operate with discrete virtual time stamps, which is significantly less natural for live streams and makes synchronization difficult. Also, Stampede requires distributed upkeep of system-wide global virtual time minimum bounds in order to perform garbage collection [151], and channels store distributed state on behalf of clients, making transparent relocation and replication difficult. *Temporal streams* uses currency bounds which require only local information. Stampede also does not provide inter-stream synchronization facilities. Finally, Stampede's programming model, like basic MPI, assumes a static participant set, which makes adapting to more widely-distributed and dynamic environments difficult. D-Stampede eliminates this limitation but at the expense of dynamic participants not being "first-class" entities (i.e., operations of dynamic participants must be proxied by a static participant). In concert with D-Stampede, Crest [30] supports reasoning about events using historical information in the context of pervasive computing applications. Crest defines a three dimensional tuple-space for historical events based on time, location and identity and provides a set of query operations to find events lying along intersecting lines or planes.

**Distributed Data Structures & Derived Storage:** As mentioned earlier, Distributed Data Structures [95] and Boxwood [132], provide various data structures as distributed programming primitives. *Distributed hash tables* (DHTs) are also a popular form of distributed data structure for distributed programming, typically presenting familiar key-value store interfaces.[8] Chord [187] (plus DHash [69], a basic block storage layer on top of

[8] Although some solutions, such as Chord [187], decouple the basic hash lookup primitive from any higher-level value storage functionality built above.

the Chord hashing primitive), CAN [168], Pastry [175] and Tapestry [211] are all notable examples. Many of these systems have been used beneath higher-level distributed-storage solutions, including various peer-to-peer filesystems. CFS [69] provides a read-only filesystem interface built on top of DHash/Chord. DHash and Chord were later used to implement Ivy [145], a peer-to-peer read-write filesystem. Similarly, both PAST [174], a peer-to-peer data publishing/archival system (with immutable data; not a read-write filesystem) and Pastiche [67], a cooperative-storage based backup system, are layered on Pastry. OceanStore [115], another wide-area distributed storage system, is itself built upon Tapestry. Note that these peer-to-peer filesystems present a filesystem-like interface and use distributed data structures underneath as an implementation mechanism, while *temporal streams* presents a distributed data structure interface and uses a storage mechanism underneath to provide persistence of data structure contents.

## *10.5 Parallel Batch Processing*

Distributed programming models and runtime systems designed for parallel processing/mining large amounts of data, such as MapReduce [71] and Dryad [107], often have similar concerns as live stream analysis applications, which makes many related techniques relevant to our domain. For example, Sawzall [163] provides a small domain-specific language for item-at-a-time processing of stored data sets within MapReduce, but it could also apply to streaming data. The key difference is that live stream analysis applications are continuous and data is explicitly time-related, while these aforementioned systems operate on stored data. Although stored data is often streamed for processing, the time at which a streamed data item becomes available for processing is unrelated to the data itself. Also,

such systems are generally optimized for throughput over latency, are not limited to one-pass processing, and typically use foreknowledge of the size of a dataset to partition processing. Operating on stored data enables certain optimizations – for example, MapReduce simply re-executes failed tasks and also uses a similar strategy when a slow mapping task is holding up progress. In fact, Isard et al. explicitly note the unsuitability of the system for such tasks: "Dryad is a batch computation system, not designed to support real-time operation which is crucial for [Continuous Query] systems since many CQ window operators depend on real-time behavior" [107].

While parallel batch computation systems such as MapReduce [71] and Dryad [107] define programming models for conveniently expressing instances of data-oriented batch computation, He et al. [102] have recently proposed *wave computing* as a new computing model encompassing the use of these batch processing systems in a larger context. These systems are often used for preprocessing indices or exploratory data processing on datasets that evolve (e.g., the entire web or sets of log files). Consequently, batch jobs are often re-run regularly to reflect changes in the input datasets. Additionally, during exploratory data processing – for example, the authors use "log mining" as a motivating scenario – several interested parties may run independent instances of MapReduce over the same dataset to extract different information. The wave computing model is designed to reflect recurrent instances of batch computation on evolving datasets. It also attempts to capture and exploit temporal relationships between multiple concurrent but normally independent instances of batch computation, allowing independent but similar MapReduce jobs to eliminate redundant computation and I/O. The authors describe wave computing as a middle-ground between stream processing and batch processing [102]. In practice, however, the data processing characteristics are somewhat incongruous. Unlike live stream processing, wave

computing deals with large finite datasets (e.g., the entire web), where the system can access a full snapshot for processing.

## *10.6 Storage*

Since *temporal streams* includes a persistence interface, some storage-related work is relevant in that context. Although much prior work could be used by our system as part of an implementation (addressing issues related to stream persistence), in the interest of brevity we will only highlight a few representative examples and one particularly relevant project: Hyperion [75] is a system for online indexing and retrieval of high-volume data streams, with network monitoring as the primary example. Although the motivation of Hyperion is supporting a streaming database style approach like GigaScope [68] (with indexing like MIND [125]), the primary contribution of Hyperion is a high-performance filesystem optimized for storing several concurrent high-bandwidth streams on a single physical disk – StreamFS.[9]

At the lowest level of the storage stack, the ATA/ATAPI-7 interface standard [19] (for storage devices such as hard drives) specifies a "streaming" command-set designed to support time-critical streaming data transfers for applications like video recording and playback, prioritizing data timeliness over integrity. *Active Disks* [171], an architecture for offloading data-centric computation to disk-resident processors (closer to the data), is realized with a stream-based programming model [29]. The programming model supports local, inline filtering and transformation of data as it is streamed from disk storage to a requesting host processor. Similarly, there is a whole class of research in *external memory* algorithms [200] where considerations of I/O performance bottlenecks between external

[9]Note: several other unrelated filesystems also use this name.

storage and internal memory are foremost – such algorithms typically process data in a streaming fashion, preferring full passes or *scans* over data rather than random access. Parallel batch processing systems such as MapReduce [71] also operate in this manner to optimize for the nature of magnetic disks.

*Temporal streams* uses time as an indexing mechanism for streaming data storage. Other, non-streaming storage systems also recognize the utility of timestamps as a fundamental indexing mechanism. Google's Bigtable [64], a distributed storage system for structured data often used with MapReduce [71], augments the traditional row/column data model with the additional fundamental dimension of time. These timestamps reflect the periodically-updated nature of the datasets Bigtable is designed to store; the *wave computing* [102] model is built around this property.

## *10.7 In Context*

The relationship between various stream databases, stream processing engines, event stream processing systems, stream languages, stream programming systems and distributed programming systems is very complex. Many systems defy categorization and there is significant overlap between categories. There are many general recurring themes woven through significantly different works in disparate areas – spatial locality, temporal sequencing, computation involved in bulk data transformation and data flow between distinct computational components at various granularities. Here we will attempt to roughly contrast a few different categories and sub-categories at a high level.

**Synchronous vs. Asynchronous:** Newton et al., the authors of WaveScript/XStream,

characterize synchrony versus asynchrony as a "major divide in [stream processing] systems" [149]. They note that the difference in approaches has mostly split on community boundaries, with database literature focusing on asynchronous models and compiler and programming language research focusing on synchronous models. While synchronous models are better for analysis and optimization, they are ultimately too restrictive for practical, large-scale live stream analysis applications. Consequently, solutions inspired by synchronous dataflow models such as StreamIt [193] and WaveScript [149] have all added support for selective asynchrony in various forms.

**Stream Programming vs. Continuous Stream Languages:** Both stream programming solutions like Brook [57] and continuous stream languages like StreamIt [193] and WaveScript [93] have adopted the terms "stream" and "streaming." Some may find a distinction between categories arbitrary; indeed both classes of solutions have ended up evolving to increasingly similar computational models applied to achieve similar goals. From the perspective of *temporal streams*, however, the distinction between the classes of solutions is their model of streams – primarily whether time is significant. Continuous stream languages generally ascribe significance to time and the rate of data arrival. In contrast, stream programming systems generally treat streams as finite sets of bulk data to process; the rate of data arrival is not critical. Gummaraju et al. [96] describe *stream programming* as follows:

> Stream programming advocates a style of programming where data is encapsulated into contiguous array of records called *streams* which get operated on by a series of computation *kernels*. ... Essentially, the program is structured in a data-flow style at the granularity of streams and kernels.

As the above definition notes, these systems generally view streams as arrays of records that need to be processed rather than as sequences of items arriving over time as they are produced. Mattson and Lethin [137] also distinguish the two categories based on whether streams are finite (as in stream programming), or infinite (as in continuous stream programming). These solutions are also distinguished in their originating research communities. Many stream programming systems were developed to scale out data-intensive calculation to utilize massively parallel hardware resources like GPUs. Mattson and Lethin [137] note that one of the main differences is the communities: Brook [57] and related solutions are "designed by application and architecture people," while StreamIt [193] and similar systems are "designed by compiler people." This distinction is now disappearing as the two models increasingly adopt similar features.

**Parallel Batch Processing vs. Stream Programming:** Parallel batch processing solutions such as MapReduce [71] and stream programming solutions such as Sequoia [83] and Streamwave [96] can be viewed as performing similar functions at significantly different levels of scale. MapReduce takes arrays of records and applies item-wise and reduction computational kernels. While MapReduce allows a single item-wise mapping and a single reduction per pass, Dryad [107] allows more general dataflow graphs. These operations look like kernel stream dataflow graphs in languages like Streamwave [96]. Fundamentally, the difference is in the level of scale and the performance goals of the system. Both types of systems attempt to take large amounts of data and scale the granularity of computation to use parallel hardware resources. Stream programming solutions typically try to optimize single-node computation where data fits in memory – optimizing the use of low-level resources of stream processors and the utilization of complex memory hierarchies by carefully orchestrating dataflow. Parallel batch processing solutions typically target clusters

of workstations and datasets too large to fit in memory – optimizing for sequential access patterns to external storage and for placement of computation near data. Gummaraju et al. [96] note the similarity:

> These efforts are mainly focused on multi-node configurations where each node could be a multi-core processor. While Streamware uses some similar techniques (e.g., for load balancing) it is distinguished by focusing on optimizing single node performance by effectively utilizing the on-chip caches and SIMD execution units of the multi-core processors using the stream programming model.

**CEP vs. Stream Databases:** Stream databases and Complex Event Processing (CEP) or Event Stream Processing (ESP) systems are very closely related and often interchangeable. One perceptible difference is in the originating research sub-community and terminology. Additionally, CEP systems are often targeted towards more specific application scenarios and workloads; consequently, the expressive power of query languages may be explicitly limited for performance reasons [9]. Demers et al. [74] describe the distinction in terms of target workload and related system design properties:

> Event processing differs from general data stream management in two major aspects of the query workload. First, it has a distinct class of queries, which warrants special attention. In complex event processing, users are interested in finding matches to event patterns, which are usually sequences of correlated events. ...
>
> Second, in complex event processing, there is usually a large number of concurrent queries registered in the event processing system. This is similar to

> the workload of publish/subscribe systems. In comparison, data stream management systems are usually less scalable in the number of queries, capable of supporting only a small number of concurrent queries. ...
>
> [Stream databases] have very powerful query languages, typically subsuming SQL with provisions for sliding windows stream grouping features. Though powerful, these query languages can be awkward for expressing the kinds of sequential patterns that occur frequently in our target applications. Moreover, the systems have yet to demonstrate the scalability of the other approaches.

**Transparent Computation vs. Opaque Computation:** In *temporal streams*, stream computation is largely opaque to the system; the programming model fundamentally concerns stream data interactions. Systems from the database and compiler communities typically have very different goals, using domain-specific knowledge of stream computation to optimize in various ways. Compiler-oriented research typified by solutions such as StreamIt [193] or WaveScript [93] tend to use global program knowledge to optimize heavily. Mattson and Lethin [137] describe the scope of streaming compilers as "[laying] out all computation, data, and communication" with applications that are "transparent to [the] compiler." Stream databases and processing engines typically optimize query plans by understanding properties of operators (like commutativity, etc.) and dynamically monitoring dataflow interactions. To optimize in this manner, such systems assume the entire application will function in a common runtime execution environment responsible for dynamically scheduling operators and orchestrating communication.

As a consequence of this focus on optimization, both database-oriented and compiler-oriented solutions tend to impose tighter coupling and more "closed world" assumptions

on applications. Many optimization techniques do not work in the face of opaque user-provided database operators or analysis code written in a common, legacy, non-stream programming language like C. In contrast, *temporal streams* targets more loosely coupled systems with opaque computation, forgoing the potential for heavy optimization in tightly coupled applications.

**Unstructured vs. Structured Streams:** *Temporal streams* is focused on the subset of live stream analysis applications involving heavyweight input streams of relatively unstructured data (such as video and audio) where the process of feature detection and analysis is the major computational requirement. Unstructured streams are data rich but are not directly interpretable until meaningful features are extracted – this area is what the system is best targeted towards and most of our application experience falls in this category. In contrast, applications with a large number of relatively lightweight streams[10] of structured data (where the individual data items are directly interpretable, like stock prices or position data), typified in Linear Road [39] and financial analysis scenarios, are often well-suited to stream database or processing engine style systems.

**Granularity of Stream Data:** There is also a wide range in the granularity of data items processed by stream systems. Some ESP/CEP systems and synchronous-dataflow based compiler-oriented systems are designed to process hundreds of thousands of tiny, individual data samples per second [93]. Database-oriented solutions and *temporal streams* tend to work with larger chunks of data and WaveScript [93] presents a middle-ground in the form of windows of isochronous data.

**Granularity of Dataflow:** In addition to variations in stream data granularity, various

[10]Individually lightweight – in aggregate the streams may constitute a large amount of data.

categories of related work model dataflow at significantly different levels. For example, *temporal streams* models coarser-grained data flowing between independent, typically distributed components. On the other hand, continuous stream languages like StreamIt [193] model dataflow between many small filters. For example, a reference JPEG decoder expressed in StreamIt consists of 66 non-identity filter instances [193], whereas *temporal streams* might model the entire JPEG decoder as a single, atomic entity (or just one step of a larger piece of analysis code). Like continuous stream languages, stream programming languages tend to model dataflow at a very fine level of granularity, in order to expose fine-grained parallelism for stream processors and similar hardware. Stream databases tend to model dataflow at a moderate granularity, occupying a middle-ground between compiler-oriented solutions and *temporal streams*.

**Streams and Computation:** Stream processing solutions specifically designed for distributed execution can be compared in two broad dimensions – their model of computation and their model of live streams (i.e., stream dataflow). Figure 82 plots several solutions or solution classes on a two-dimensional graph corresponding to these properties. Although condensing entire systems to a simplistic two figure classification is obviously very imprecise and somewhat subjective, it is suitable for illustrative purposes. In the bottom left corner, we have general distributed programming and communication solutions. These systems provide for distributed messaging but are not specifically targeted towards live stream-oriented applications. In the upper right-hand corner, we have a series of declarative stream processing solutions, which may have SQL-like continuous query languages for expressing stream computation. Stream computation and stream dataflow are all implicitly supported in a domain-tailored manner. SPADE [88] would fall into this category, but the Stream Processing Core [34] without SPADE on top supports lower-level, less declarative

computation. SPC's model of streams is still a fairly expressive publish/subscribe-inspired model, however. Finally, at the upper left-hand corner we have *temporal streams*, which provides a time-based model for stream data interactions but avoids modeling most stream computation. The one limited way in which *temporal streams* models stream computation is through pickling handlers, which define mapping functions executed on stream data transitioning to secondary storage. As we have discussed in detail in this chapter, all of these points in the design space are potentially useful for particular classes of applications and entail certain tradeoffs.

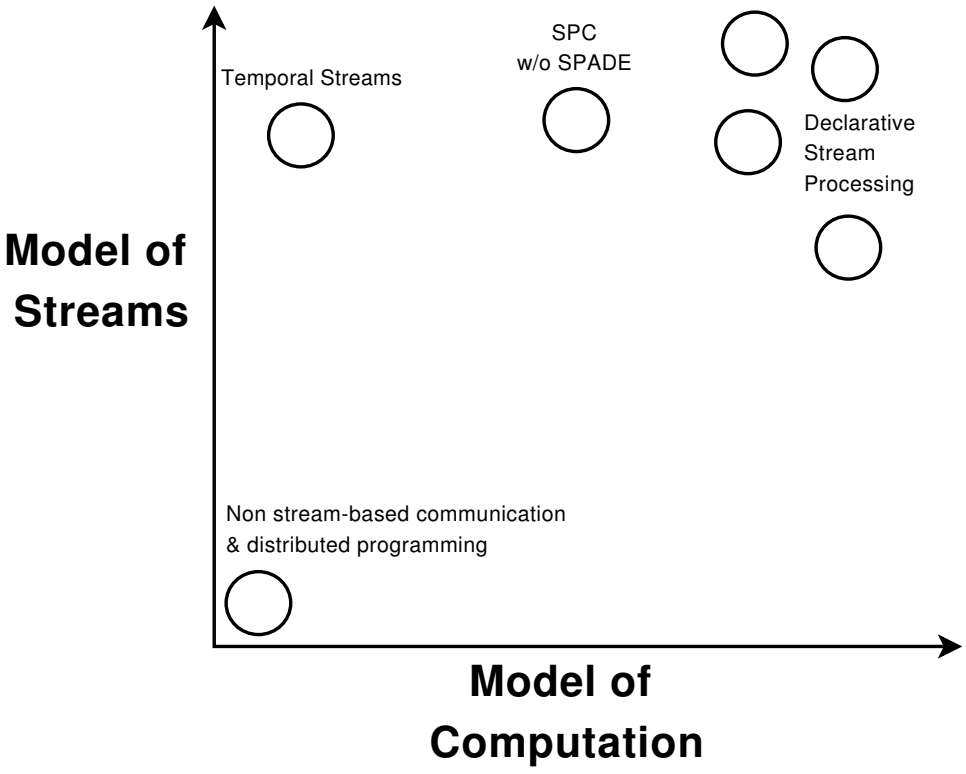



**Figure 82:** Model of streams and model of computation

Ultimately the concept of a *stream* is a very general theme in computer science, as is the concept of *dataflow*. Processing potentially unbounded sequences of data as they are generated is fundamental, both for flexibility and for performance – sequential access is often faster due either to properties of external storage or to locality of reference assumptions made at all levels of hardware. Many infinite streams are inherently related to time, which is another general theme underlying many systems in different areas. In "Computing Needs Time" [120], Edward A. Lee argues that when computers interact with physical processes, the concept of time is so essential that the core abstractions of computing at every level must be fundamentally redesigned – from micro-architecture to programming languages to operating systems to networks.

# CHAPTER XI

# CONCLUSION & FUTURE WORK

> "Perfection is achieved not when there is nothing more to add, but when there is nothing more to remove."
>
> – Antoine de Saint-Exupéry, Terre des Hommes

## *11.1 Conclusion*

A large number of future critical applications will involve continuous and computationally intensive analysis on live data streams. This work presents *temporal streams*, a programming model designed to address certain complexities inherent in constructing distributed *live stream analysis* applications. *Temporal streams* acts as a lightweight glue between communicating application components and provides a simple but flexible time-oriented stream primitive. This abstraction, called a *channel*, encompasses both communication and storage spanning live and historical stream data thus presenting a unified stream *data* abstraction. The time-based data indexing used by *channels* is natural for continuous data streams and provides a fundamental mechanism to synchronize data from multiple sources and reason about temporal correspondences. The quote at the beginning of this section exemplifies the philosophy shaping the *temporal streams* programming model – our goal is providing the simplest primitive high-level enough to capture the essence of a particular problem while being low-level enough to be small, efficient and widely applicable.

In addition to developing the abstract programming model, we have also examined the

issues inherent in efficient realization of *temporal streams* as a distributed runtime, using the programming model as a vehicle for exploring systems software design issues. We describe both the general distributed architecture of our runtime as well as the concrete software architecture as implemented in two generations of prototypes. Using both targeted system-level benchmarks and application-based evaluations, we have analyzed the performance of our system prototype in detail. In addition, we look at elements of each application's implementation using *temporal streams* and compare their requirements. We conclude by summarizing several potential avenues of future work for *temporal streams*.

## *11.2 Future Work*

Broadly, directions for future work can apply at different levels – we can evolve the abstract, high-level *temporal streams* programming model, or we can pursue enhanced features in the system design and implementation. In addition, future work involving specific application studies might point to potential improvements in both. Here we will summarize a few different avenues of exploration covering all system levels.

**Reverse Time Variables:** In Section 8.4 (under "Enhancements"), we noted how the traffic monitoring scenario could benefit from programming model support for reverse time intervals. In retrospect, this functionality seems obvious and is trivial to support in existing implementations. In particular, we suggest the addition of time variables to specify time intervals backward from existing items. On possibility is *previous*, which is analogous to *next* but operating in the opposite direction. The second and more general possibility is *n-before*, which allows reverse navigation in a channel by a certain number of items.[1] Currently, reverse temporal navigation in channels can be performed by concrete time amounts

[1] Technically *previous* is just *n-before* when $n = 1$.

(e.g., 10 ms. earlier), but *n-before* allows this navigation to depend on the timestamps of items in the channel.

These possibilities also point to other potential time variables to add; specifically, our set of forward and reverse variables should be symmetric if possible. For example, *n-before* implies an analogous *n-after* which is like *next* but allows forward navigation of *n* items rather than a single item. The existence of *newest-after*(ts) implies some sort of predicated equivalent such as *oldest-before*(ts), which would be equal to *oldest* if ts is newer than *oldest* and $\infty$ otherwise. It is not clear intuitively when *oldest-before* would be needed, so the suggestion is only based on the desire for symmetry in the set of defined time variables. While symmetry is appealing, future experiences with new application domains are superior in evincing time variable deficiencies to rectify.

**Auto-Sizing of Live Windows:** In Section 3.6, we discuss channel garbage collection in the programming model. Typically, the programmer specifies a time-based window of live data to keep in the channel in memory; for example, a video channel might keep the most recent five minutes of data. In this manner, users are implying that data with a certain level of currency is important. In Section 3.6 we also mention the possibility of dynamically adjusting these windows based on factors such as the number of consumers or estimated end-to-end latency. Using the enhanced stream metadata facilities discussed in Section 6.7.2 for stream priority and feedback, the system could dynamically increase the amount of data kept in memory for streams that are currently "important" based on priority information or based on feedback from analysis components in danger of temporarily falling behind. Allowing the system to dynamically adapt based on application-provided information helps to improve overall performance and quality of service.

**Providing Hooks for Upper Layers:** Both Section 6.7 and Section 8.5 discuss the nature of additions to *temporal streams* in light of its intended purpose and design philosophy. In particular, *temporal streams* should provide hooks to effectively support a variety of higher-level domain-specific functionality while remaining a flexible lower-level substrate. It should only provide core mechanisms which are widely applicable or cannot be properly implemented at a higher level. Ultimately, system-level hooks provide opportunities for bidirectional information exchange between an application and the underlying *temporal streams* layer, allowing both layers to make better decisions. Performance information provided by *temporal streams* to the application can allow it to adapt more effectively, and information provided by the application to *temporal streams* can take advantage of domain-specific knowledge to more accurately assess properties like stream priority. This bidirectional information exchange is particularly important when the "application" directly above *temporal streams* is actually an extra layer of middleware. Each extra system layer abstracts and hides information; smart cross-layer sharing need not be overly complex to ameliorate some of the downsides of extra layering.

The aforementioned feedback mechanism to control the size of channels' live windows is one form of system hook provided to upper layers to allow application-directed adaptation – this example allows the system use application-provided feedback. In the other direction, the system can provide introspective system performance information to the application. Ruan and Pai [176] demonstrate the significant effectiveness of this technique in the context of operating systems providing system call performance information to server applications, but their general lesson can hold for many kinds of multi-layer systems. *Temporal streams* could provide optional introspective performance information on channel *get* and *put* calls. Solaris's dynamic tracing/instrumentation framework DTrace [60] is often

used in a similar capacity to great success. Although these tools frequently focus on tracing and instrumentation of production applications for performance optimization or troubleshooting, such information could also be used for automatic dynamic adaptation (e.g., load shedding).

**Implementation-Directed Work:** Section 6.7 covers extensive potential future enhancements primarily focused at the level of concrete system implementation. The section discusses an HTTP-based wire protocol (Section 6.7.1), enhanced stream naming and metadata facilities (Section 6.7.2), enhanced persistent data lifecycle management (Section 6.7.3), security considerations (Section 6.7.4) and miscellaneous straightforward implementation enhancements (Section 6.7.5). Of these improvements, the enhanced stream naming and metadata facilities, the enhanced persistent data lifecycle management and security considerations could also be visible at the higher level of the programming model. In particular, enhanced stream "subscription" features or dynamic stream "views" could be defined in the programming model. Some stream processing systems, such as Media Broker [144] also define built-in type systems.

Enhanced persistent data lifecycle management could also change the view of channels presented by the programming model. Currently, each channel presents a simple linear process: live data enters a channel via *put* calls and stays in the channel until it is old enough to be garbage collected. After that, the data is either reclaimed or sent to the persistence layer, which may optionally transform the data with pickling handlers. With a more advanced persistent data lifecycle, the data may be transformed multiple times by pickling handlers with different tradeoffs. In addition, with increasing interest in solid-state storage, the difference in access times between "live" and first-level historical data may not be as stark as with traditional magnetic media. Finally, aspects of security and access

control could be visible at the programming model level. For example, the E programming language defines a powerful capability model for distributed programming [142].

**Future Application Work:** In Chapter 8, we evaluate *temporal streams* using three concrete, motivating application scenarios. Section 8.5 describes *Video as a Service*, a new application using *temporal streams*. Future application studies provide the opportunity for symbiotic enhancement – *temporal streams* provides a basic substrate for live stream analysis and new application domains inevitably point to improvements in the programming model or implementation. We have already provided one in-depth example: richer temporal semantics for reverse time intervals. Besides *Video as a Service*, we are also looking at applying *temporal streams* in the context of an IPTV content recommender system and the RF$^2$ID [33] system for cargo tracking applications using RFID.

**Broad Programming Model Improvements:** In addition to our aforementioned directed programming model improvements (developed in response to immediate application needs), we also have germinal ideas for broader evolution. One specific direction is developing an expressive model for keeping the application "in the loop" to handle application-level failures that the system detects. As the scope of applications grows ever larger, both in the amount of data processed and the number of concurrently executing tasks, failure probabilities of individual components compound to the point where they can no longer be ignored. Live streaming applications in particular are continuously running applications, so fail-stop error handling is unacceptable. Below the application, the system can detect and mitigate certain types of failures through standard techniques like replication, monitoring and migration.

Some types of failures, however, must be exposed to the application: for example, if

an un-replicated sensor fails and stops producing data, the application must be notified to properly handle the condition. This kind of failure can also result in problems when *temporal streams* synchronization is used – if a sensor stops producing data, synchronized data sources may also stop (waiting for newer data). A "missing data" exception mechanism could address the synchronization and notification scenario, but many other types of failures must also be considered. In some cases it might be appropriate to replay old readings or extrapolate missing data by using techniques like *dead reckoning*. Other potential topics for broad programming model augmentation include dealing with power concerns, and handing trust or confidence in individual feature detectors results or data sources coming from multiple parties (see Section 6.7.4).

**Novel System Architectures:** In the process of developing a native Java re-implementation of the *temporal streams* runtime, I thought about embedding *temporal streams* in a filesystem implementation. Although the concept of exposing middleware through lowest-common-denominator kernel interfaces might be considered an inelegant "hack," the subsequent proliferation of novel FUSE-based (Filesystem in Userspace) [10] filesystems for increasingly diverse tasks has demonstrated that the idea is potentially practical, even if inelegant by some measures. As mentioned in Section 6.6.3, the parallel re-implementation of *temporal streams* in Java is motivated by the desire to avoid problems associated with JNI. An alternate strategy is to provide the *temporal streams* functionality through the kernel filesystem interface. Since all viable languages can make system calls and have support for reading files, this is a practical strategy for making *temporal streams* functionality available to systems in many languages with less effort. In a filesystem-based implementation, *get* operations map to `read` calls and *put* operations map to `write` calls. Timestamp and

time interval information can be encoded into file offsets used in `lseek` or, preferably, directly to calls like `pread` and `pwrite`. Channels could be created and manipulated using operations like `open` and possibly `ioctl`. Although this interface is ugly and austere, it could be wrapped in a lightweight, language-specific facade; the majority of the system's functionality could then be used in each language without the need for re-implementation or using general-purpose native code interfaces like JNI.

Although implementation convenience was my original motive, the thought experiment of mapping *temporal streams* to a filesystem-based interface prompted me to consider the similarities between distributed filesystems and stream data abstractions in live stream analysis applications. There are many significant similarities between distributed filesystems and *temporal streams* – fundamentally, both present named read/write structured data abstractions to distributed applications. Mapping live streams onto files in a distributed filesystem is remarkably suitable. One superficial difference is the typical file system's use of byte-addressing versus the item-based (and time-indexed) nature of streams in *temporal streams*. The concept of a record-oriented filesystem is an old idea common in mainframe systems, but modern filesystems uniformly tend to favor byte-indexed filesystems. The idea is coming back, however, in new filesystems like DragonFlyBSD's HAMMER [77]. One of Matthew Dillon's original HAMMER design documents states the following:

> HAMMER uses 64 bit keys internally and makes key-based files directly available to userland. Key-based files are not regular files and do not operate using a normal data offset space. ...
>
> Reads which normally seek the file forward will instead iterate through the records and `lseek`/`qseek` can be used to acquire or set the key prior to the `read`/`write` operation.

Besides the access paradigm, live streams are conceptually infinite and constantly updated. Some filesystem properties like the size of a file are not as meaningful on live streams. Despite the fact that live streams are conceptually infinite, this distinction is somewhat unimportant as they are practically finite – *temporal streams*'s view of streams includes some recent finite "tail" of a continuous stream's data. Furthermore, the fundamental structure of a channel in *temporal streams* contains a small set of in-memory "live" data and less-recent historical data on secondary storage. This structure nicely corresponds to an operating system's in-memory buffer cache holding in-use and recently-used blocks of files fully backed on secondary storage. The correspondence becomes even closer when one compares advanced ILM features of some filesystems (like GPFS [177]) to pickling handlers[2] and advanced hierarchical *temporal streams* backends (see Section 6.7.3). The similarities between distributed filesystems and *temporal streams* are compelling; further analysis both of the similarities and fundamental differences between the two systems will likely elucidate useful optimizations and synergistic opportunities.

[2]Pickling handlers could be analogized as a stronger, domain-specific version of selective, in-filesystem lossless compression provided by some filesystems (e.g., ZFS [54]).

# APPENDIX A

# A SET-THEORETIC "CASUAL FORMALIZATION" OF CHANNELS

One possible view of a channel is a sequence of items $i_0, i_1, \ldots, i_n$ where each item $i_i$ consists of a timestamp $t_i$ and a data item $d_i$: $i_i = (t_i, d_i)$. The sequence is ordered according to ascending $t_i$ so $i_0, i_1, \ldots, i_n$ implies that $t_i \leq t_{i+1}$. We will call this view the *item-based* representation of channels. This view is close to a realistic implementation of a channel, but we can view channels in different way which is more convenient for defining certain operations. The second representation, called the *timestamp-based* representation, handles items with identical timestamps by associating a set of data items with a single timestamp.

In this alternate definition, a channel can be viewed as a sequence of timestamps: $t_0, t_1, \ldots, t_n$ where $t_i < t_{i+1}$. Each timestamp has an associated non-empty set of data items, $d_0, d_1, \ldots, d_n$ where $|d_i| > 0$. $d_i$ is a set of data items rather than a single item because duplicate items with the same timestamp may be allowed in an implementation.[1] As mentioned earlier, each channel item spans a time interval. If we define a data item $i_j$ in the timestamp-based representation as $i_j = (t_j, d_j)$, an item $i_j$'s interval is as follows ($i_n$ is the most recent item in the channel):

$$[\, t_j,\ t_{j+1}\,) \quad \text{if } j < n;$$

$$[\, t_j,\ now\,) \quad \text{if } j = n.$$

[1] We can map the timestamp-based representation to the original item-based representation: first we assign an arbitrary order to each set $d_i$ so $d_i = \{d_{(i,0)}, d_{(i,1)}, \ldots\}$. Then our sequence of items is simply $(t_o, d_{0,0}),\ (t_0, d_{0,1}), \ldots,\ (t_1, d_{1,0}), \ldots$.

We'll define upper($t_i$) to be the upper bound of the above interval ($t_{i+1}$ or *now* if no such successor exists). The basic get function takes an interval in the form of lower and upper bound timestamps ($l \leq u$) and can be defined as follows:

$$\text{get}(l, u) = \{d \mid d \in d_i, \forall i \text{ where } (l \leq t_i \wedge u > t_i)\}$$

Basically, get returns all individual data items $d_i$ where each $d_i$'s corresponding interval, $[t_i, \text{upper}(t_i))$, intersects the interval $[l, u)$, starting with the first fully contained item. Replacing the second clause of the conjunction with $u \geq \text{upper}(t_i)$ gives only fully contained items.

As mentioned earlier, a channel group simply modifies the visibility of channel items to provide the illusion of related items becoming available simultaneously. A channel group $g$ can be defined as a pair $g = (c_g, c_r)$ consisting of a set of $n$ channels $c_g = c_0, c_1, \ldots, c_n$ and a reference channel $c_r$ which is either some $c \in c_g$ or the special variable *oldest*. First we define the newest timestamp in a channel:

$$\text{newest}(c) = \{t_i \in c \mid \text{upper}(t_i) = \textit{now}\}$$

A channel group's *frontier timestamp* $t_g$ is defined as follows:

$$\begin{array}{ll} \min(t_i \in S) \text{ where } S = \{\text{newest}(c),\ \forall c \in c_g\} & \text{if } c_r = \textit{oldest}; \\ \text{newest}(c_r) & \text{otherwise.} \end{array}$$

A get on a channel modified by a channel group $g$ behaves as if all $t_i > t_g$ do not exist. Time variables are also interpreted with respect to the reference stream $c_r$.

# REFERENCES


[1] "Apache Etch." http://cwiki.apache.org/ETCH/. Accessed: April 2009. 6.6.2

[2] "Apache Hadoop." http://hadoop.apache.org/. Accessed: March 2009. 6.5, 14

[3] "Apache ZooKeeper." http://hadoop.apache.org/zookeeper/. Accessed: March 2009. 6.5, 14, 25, 6.7.4

[4] "btrfs Wiki." http://btrfs.wiki.kernel.org. Accessed: January 2009. 5

[5] "Coral8 CCL Reference." Coral8 Inc. http://www.coral8.com/system/files/assets/pdf/5.2.0/Coral8CclReference.pdf, Accessed: January 2009. 10.1

[6] "Coral8 Inc.." http://www.coral8.com, Accessed: January 2009. 10.1

[7] "CouchDB – Frequently asked questions." http://wiki.apache.org/couchdb/Frequently_asked_questions. Accessed: March 2009. 6.5

[8] "Drools : Business Logic integration Platform." http://www.jboss.org/drools/. Accessed: May 2009. 8.2, 8.2.3

[9] "Expressiveness of Stream Processing Languages." Cornell University Database Systems (research initiative). http://www.cs.cornell.edu/bigreddata/cayuga/expressiveness/, Accessed: January 2009. 10.1, 10.7

[10] "FUSE: Filesystem in Userspace." http://fuse.sf.net. Accessed: June 2009. 11.2

[11] "Java Remote Method Invocation – Distributed Computing for Java." Sun Microsystems Whitepaper. http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp, Accessed: January 2009. 2.3, 10.4

[12] "Middleware for Large-Scale Surveillance (MILS)." IBM Research. http://domino.research.ibm.com/comm/research_projects.nsf/pages/s3.mils.html, Accessed: January 2009. 10.1

[13] "Netty - The Java NIO Client Server Socket Framework." http://www.jboss.org/netty/index.html. Accessed: March 2009. 6.5, 6.6.2

[14] "OpenCV Wiki." http://opencv.willowgarage.com/wiki/, Accessed: May 2009. 8.2, 8.2.1, 8.4

[15] "Protocol Buffers - Google's data interchange format." http://code.google.com/p/protobuf/. Accessed: March 2009. 6.5, 6.6.2, 6.6.2, 8.3.1

[16] "ruleCore CEP Server." http://rulecore.com. Accessed: January 2009. 10.1

[17] "StreamBase Streaming Platform." http://www.streambase.com/, Accessed: January 2009. 10.1

[18] "Truviso." http://www.truviso.com. Accessed: January 2009. 10.1

[19] "AT Attachment with Packet Interface - 7 (ATA/ATAPI-7) – Volume 1: Register Delivered Command Set, Logical Register Set." ANSI INCITS Working Draft T13 1532D Volume 1 (NCITS 397-2005), March 2004. Revision 4a. 10.6

[20] "ISO/IEC 9834-8: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components," 2005. ITU-T Recommendation X.667. 6.3.1

[21] "The PeakStream Platform: High Productivity Software Development for Multi-core Processors." PeakStream Inc., 2006. 10.2.1

[22] "StreamSQL online documentation." StreamBase Systems, 2007. http://streambase.com/developers/docs/latest/streamsql/index.html, Accessed: January 2009. 10.1

[23] "Esper Reference Documentation," 2008. http://esper.codehaus.org/, Accessed: January 2009. 10.1

[24] "IEEE 1588-2008 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems." Institute of Electrical and Electronics Engineers, 2008. 3.3

[25] "Real-Time Media Flow Protocol: Frequently Asked Questions – External." Adobe Labs FAQ, November 2008. 10.4

[26] A. Courtney and W. Hanssen, et al., "Inter-Language Unification, Release 1.5," Tech. Rep. ISTL-CSA94-01-01, Xerox PARC, May 1994. 6.6.2

[27] Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S., "The Design of the Borealis Stream Processing Engine," in *Proceedings*

*of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR '05)*, (Asilomar, CA), January 2005. 6.6.4, 8.5, 10.1

[28] Abelson, H. and Sussman, G. J., *Structure and Interpretation of Computer Programs (SICP)*. Cambridge, MA, USA: MIT Press, 1996. 3.5 – Streams. 10.3

[29] Acharya, A., Uysal, M., and Saltz, J., "Active Disks: Programming Model, Algorithms and Evaluation," *ACM SIGPLAN Notices*, vol. 33, no. 11, pp. 81–91, 1998. 10.6

[30] Adhikari, S., *Programming Idioms and Runtime Mechanisms for Distributed Pervasive Computing*. PhD thesis, College of Computing, Georgia Institute of Technology, October 2004. 9, 10.4

[31] Adhikari, S., Paul, A., and Ramachandran, U., "D–Stampede: Distributed Programming System for Ubiquitous Computing," in *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS '02)*, pp. 209–216, July 2002. 2.3, 6.6.2, 10.4

[32] Agarwalla, B., Ahmed, N., Hilley, D., and Ramachandran, U., "Streamline: Scheduling Streaming Applications in a Wide Area Environment," *Multimedia Systems (MMSJ)*, vol. 13, pp. 69–85, September 2007. 2.3

[33] Ahmed, N., Kumar, R., French, R. S., and Ramachandran, U., "RF$^2$ID: A Reliable Middleware Framework for RFID Deployment," in *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS '07)*, pp. 1–10, March 2007. 2.3, 11.2

[34] Amini, L., Andrade, H., Bhagwan, R., Eskesen, F., King, R., Selo, P., Park, Y., and Venkatramani, C., "SPC: A Distributed, Scalable Platform for Data Mining," in *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms (DMSSP '06)*, (New York, NY, USA), pp. 27–37, ACM, 2006. 6.6.4, 6.7.2, 10.1, 10.4, 10.7

[35] Anderson, R. and Needham, R., *Programming Satan's Computer*, vol. 1000 of *Lecture Notes in Computer Science*, pp. 426–440. Springer Berlin / Heidelberg, 1995. 26

[36] Appavoo, J., Uhlig, V., and Waterland, A., "Project Kittyhawk: Building a Global-Scale Computer: Blue Gene/P as a Generic Computing Platform," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 1, pp. 77–84, 2008. 2

[37] Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., and Widom, J., "STREAM: The Stanford Data Stream Management System," in *Data Stream Management: Processing High-Speed Data Streams* (Garofalakis, M., Gehrke, J., and Rastogi, R., eds.), Springer, August 2008. *(in press)*. 10.1

[38] Arasu, A., Babu, S., and Widom, J., "The CQL Continuous Query Language: Semantic Foundations and Query Execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006. 10.1

[39] Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A. S., Ryvkina, E., Stonebraker, M., and Tibbetts, R., "Linear Road: A Stream Data Management Benchmark," in *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)*, pp. 480–491, VLDB Endowment, 2004. 1, 10.1, 10.7

[40] Arvind, Gostelow, K. P., and Plouffe, W. E., "An asynchronous programming language and computing machine," Tech. Rep. 114a, Department of Information and Computer Science, University of California, Irvine, 1978. 10.3

[41] Ashcroft, E. A. and Wadge, W. W., "Lucid, a Nonprocedural Language with Iteration," *Communications of the ACM*, vol. 20, no. 7, pp. 519–526, 1977. 10.3

[42] Bala, V., Bruck, J., Cypher, R., Elustando, P., Ho, A., Ho, C.-T., Kipnis, S., and Snir, M., "CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 2, pp. 154–164, 1995. 10.4

[43] Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Galvez, E., Salz, J., Stonebraker, M., Tatbul, N., Tibbetts, R., and Zdonik, S., "Retrospective on Aurora," *The VLDB Journal*, vol. 13, no. 4, pp. 370–383, 2004. 10.1

[44] Banavar, G., Ch, T., Strom, R., and Sturman, D., "A Case for Message Oriented Middleware," in *Proceedings of the 13th International Symposium on Distributed Computing (DISC '99)*, pp. 1–18, Springer-Verlag, 1999. 10.4

[45] Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R. E., and Sturman, D. C., "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems," in *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, (Washington, DC, USA), p. 262, IEEE Computer Society, 1999. 10.4

[46] Barga, R. S., Goldstein, J., Ali, M., and Hong, M., "Consistent Streaming Through Time: A Vision for Event Stream Processing," in *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR '07)*, January 2007. 10.1

[47] Baset, S. A. and Schulzrinne, H., "An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol," in *Proceedings of the 25th IEEE Conference on Computer Communications (INFOCOM '06)*, pp. 1–11, April 2006. 10.4

[48] Beazley, D. M., "SWIG: An easy to use tool for integrating scripting languages with C and C++," in *Proceedings of the 4th USENIX Tcl/Tk Workshop*, (Monterey, CA), July 1996. 6.6.3

[49] Birman, K. and Joseph, T., "Exploiting Virtual Synchrony in Distributed Systems," *ACM SIGOPS Operating System Review*, vol. 21, no. 5, pp. 123–138, 1987. 10.4

[50] Birman, K. P., "Overcoming Challenges of Maturity," in *Proceedings of the Workshop on Organizing Workshops, Conferences, and Symposia for Computer Systems (WOWCS'08)*, (Berkeley, CA, USA), pp. 1–5, USENIX Association, 2008. 9

[51] Birrell, A. D. and Nelson, B. J., "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 1984. 10.4

[52] Black, A. P., Huang, J., Koster, R., Walpole, J., and Pu, C., "Infopipes: An Abstraction for Multimedia Streaming," *Multimedia Systems*, vol. 8, no. 5, pp. 406–419, 2002. 10.4

[53] Blelloch, G. E., "NESL: A Nested Data-Parallel Language (Version 2.6)," tech. rep., Pittsburgh, PA, USA, 1993. 10.2.1

[54] Bonwick, J., Ahrens, M., Henson, V., Maybee, M., and Shellenbaum, M., "The Zettabyte File System," tech. rep., Sun Microsystems. 2

[55] Braam, P. J., "The Lustre Storage Architecture." Cluster File Systems, Inc., August 2004. 5.2

[56] Breslau, L., Cue, P., Cao, P., Fan, L., Phillips, G., and Shenker, S., "Web Caching and Zipf-like Distributions: Evidence and Implications," in *In Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '99)*, pp. 126–134, March 1999. 9

[57] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P., "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3, pp. 777–786, 2004. 10.2.1, 10.7

[58] Burns, B., Grimaldi, K., Kostadinov, A., Berger, E. D., and Corner, M. D., "Flux: A Language for Programming High-Performance Servers," in *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX '06)*, (Berkeley, CA, USA), pp. 13–13, USENIX Association, 2006. 6.6.3

[59] Burrows, M., "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, (Berkeley, CA, USA), pp. 335–350, USENIX Association, 2006. 6.5, 14, 6.6.3

[60] Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H., "Dynamic Instrumentation of Production Systems," in *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX '04)*, (Berkeley, CA, USA), USENIX Association, 2004. 11.2

[61] Carriero, N. and Gelernter, D., "Linda in context," *Communications of the ACM*, vol. 32, no. 4, pp. 444–458, 1989. 10.4

[62] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L., "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems (TOCS)*, vol. 19, no. 3, pp. 332–383, 2001. 10.4

[63] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M., "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," in *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR '03)*, January 2003. 10.1

[64] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E., "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008. 2.4, 6.5, 10.6

[65] Cipriani, N., Eissele, M., Brodt, A., Grossmann, M., and Mitschang, B., "NexusDS: A Flexible and Extensible Middleware for Distributed Stream Processing," in *Proceedings of the 2009 International Symposium on Database Engineering & Applications (IDEAS '09)*, September 2009. 10.1

[66] Consel, C., Hamdi, H., Réveillère, L., Singaravelu, L., Yu, H., and Pu, C., "Spidle: A DSL Approach to Specifying Streaming Applications," in *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (CPCE '03)*, (New York, NY, USA), pp. 1–17, Springer-Verlag New York, Inc., 2003. 10.2.2

[67] Cox, L. P., Murray, C. D., and Noble, B. D., "Pastiche: Making Backup Cheap and Easy," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 285–298, 2002. 10.4

[68] Cranor, C., Johnson, T., Spataschek, O., and Shkapenyuk, V., "Gigascope: A Stream Database for Network Applications," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*, (New York, NY, USA), pp. 647–651, ACM Press, 2003. 10.1, 10.6

[69] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I., "Wide-area cooperative storage with CFS," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*, (New York, NY, USA), pp. 202–215, ACM Press, 2001. 10.4

[70] DARPA: Information Processing Technology Office, "Polymorphous Computing Architectures (PCA)." `http://www.darpa.mil/IPTO/programs/pca/pca.asp`. Accessed: August 2009. 10.2.1

[71] Dean, J. and Ghemawat, S., "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004. 6.5, 14, 10.3, 10.5, 10.6, 10.7

[72] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W., "Dynamo: Amazon's Highly Available Key-Value Store," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, (New York, NY, USA), pp. 205–220, ACM, 2007. 2.4

[73] Dekkers, P., "Complex Event Processing," Master's thesis, Radboud University Nijmegen, October 2007. 10.1

[74] Demers, A., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., and White, W., "Cayuga: A High-Performance Event Processing Engine," in *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR '07)*, January 2007. 10.1, 10.4, 10.7

[75] Desnoyers, P. and Shenoy, P., "Hyperion: High Volume Stream Archival for Retrospective Querying," in *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX '07)*, pp. 45–58, June 2007. 5.2.1, 5.2.2, 6.2.3, 10.1, 10.6

[76] Dierks, T. and Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.2." RFC 5246 (Proposed Standard), Aug. 2008. 6.7.4

[77] Dillon, M., "THE HAMMER FILESYSTEM." http://www.dragonflybsd.org/hammer/index/hammer.pdf, June 2008. Accessed: January 2008. 5, 11.2

[78] Dongarra, J., Geist, G. A., Manchek, R., and Sunderam, V. S., "Integrated PVM Framework Supports Heterogeneous Network Computing," *Computers in Physics*, vol. 7, no. 2, pp. 166–174, 1993. 10.4

[79] Eide, E., Frei, K., Ford, B., Lepreau, J., and Lindstrom, G., "Flick: A Flexible, Optimizing IDL Compiler," *ACM SIGPLAN Notices*, vol. 32, no. 5, pp. 44–56, 1997. 6.6.2

[80] Eisler, M., "XDR: External Data Representation Standard." RFC 4506 (Standards Track), May 2006. 6.5

[81] Elson, J., Girod, L., and Estrin, D., "Fine-Grained Network Time Synchronization using Reference Broadcasts," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, (New York, NY, USA), pp. 147–163, ACM, 2002. 3.3

[82] Fagg, G. E. and Dongarra, J. J., "FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world," *Lecture Notes in Computer Science*, vol. 1908, pp. 346–353, 2000. 10.4

[83] Fatahalian, K., Horn, D. R., Knight, T. J., Leem, L., Houston, M., Park, J. Y., Erez, M., Ren, M., Aiken, A., Dally, W. J., and Hanrahan, P., "Sequoia: Programming the Memory Hierarchy," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*, (New York, NY, USA), p. 83, ACM Press, 2006. 10.2.1, 10.7

[84] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T., "Hypertext Transfer Protocol – HTTP/1.1." RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817. 6.7.1

[85] Fielding, R. T., *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. 6.4, 8.5

[86] Fitzpatrick, T., Gallop, J. J., Blair, G. S., Cooper, C., Coulson, G., Duce, D. A., and Johnson, I. J., "Design and Application of TOAST: An Adaptive Distributed Multimedia Middleware Platform," in *IDMS '01: Proceedings of the 8th International Workshop on Interactive Distributed Multimedia Systems*, (London, UK), pp. 111–123, Springer-Verlag, 2001. 10.4

[87] Forum, M. P. I., "MPI: A Message-Passing Interface Standard," Tech. Rep. UT-CS-94-230, 1994. 2.3, 10.4

[88] Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M., "SPADE: The System S Declarative Stream Processing Engine," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, (New York, NY, USA), pp. 1123–1134, ACM, 2008. 3.5, 10.1, 10.2.1, 10.7

[89] Ghemawat, S., Gobioff, H., and Leung, S.-T., "The Google File System," *SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003. 6.5, 14

[90] Ghosh, S. P. and Senko, M. E., "File Organization: On the Selection of Random Access Index Points for Sequential Files," *Journal of the ACM (JACM)*, vol. 16, no. 4, pp. 569–579, 1969. 6.2.3

[91] Ghuloum, A., Smith, T., Wu, G., Zhou, X., Fang, J., Guo, P., So, B., Rajagopalan, M., Chen, Y., and B., C., "Future-Proof Data Parallel Algorithms and Software on Intel® Multi-Core Architecture," *Intel Technology Journal*, vol. 11, pp. 333–348, November 2007. 10.2.1

[92] Ghuloum, A., "Ct: Channelling NeSL and SISAL in C++," in *Proceedings of the 4th ACM SIGPLAN Workshop on Commercial Users of Functional Programming (CUFP '07)*, (New York, NY, USA), pp. 1–3, ACM, 2007. 10.2.1

[93] Girod, L., Mei, Y., Newton, R., Rost, S., Thiagarajan, A., Balakrishnan, H., and Madden, S., "The Case for a Signal-Oriented Data Stream Management System," in *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR '07)*, (Monterey, CA), January 2007. 8.5, 10.1, 10.2.2, 10.7

[94] Girod, L., Mei, Y., Rost, S., Thiagarajan, A., Balakrishnan, H., and Madden, S., "XStream: A Signal-Oriented Data Stream Management System," in *Proceedings of the 24th International Conference on Data Engineering (ICDE '08)*, (Canc'un, M'exico), April 2008. 10.1, 10.2.1, 10.2.2

[95] Gribble, S. D., Brewer, E. A., Hellerstein, J. M., and Culler, D., "Scalable, Distributed Data Structures for Internet Service Construction," in *Proceedings of the Fourth USENIX Symposium on Operating System Design and Implementation (OSDI'00)*, (Berkeley, CA, USA), pp. 22–22, USENIX Association, 2000. 2.2, 10.1, 10.4

[96] Gummaraju, J., Coburn, J., Turner, Y., and Rosenblum, M., "Streamware: Programming General-Purpose Multicore Processors Using Streams," in *Proceedings of*

*the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, (New York, NY, USA), pp. 297–307, ACM, 2008. 10.2.1, 10.7

[97] Gyllstrom, D., Wu, E., Chae, H.-J., Diao, Y., Stahlberg, P., and Anderson, G., "SASE: Complex Event Processing over Streams," in *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR '07)*, January 2007. Demo proposal. 10.1

[98] Hamid, R., Johnson, A., Batta, S., Bobick, A., Isbell, C., and Coleman, G., "Detection and Explanation of Anomalous Activities: Representing Activities as Bags of Event n-Grams," in *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '05)*, vol. 1, (Washington, DC, USA), pp. 1031–1038, IEEE Computer Society, 2005. 8.1.3

[99] Hapner, M., Burridge, R., Sharma, R., Fialli, J., and Stout, K., "Java Message Service Specification." Sun Microsystems, April 2001. Version 1.1. 2.3, 10.4

[100] Haridi, S., Roy, P. V., Brand, P., and Schulte, C., "Programming Languages for Distributed Applications," *New Generation Computing*, vol. 16, no. 3, pp. 223–261, 1998. 10.4

[101] Harrison, T. H., Levine, D. L., and Schmidt, D. C., "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '97)*, (New York, NY, USA), pp. 184–200, ACM Press, 1997. 6.6.2

[102] He, B., Yang, M., Guo, Z., Chen, R., Lin, W., Su, B., Wang, H., and Zhou, L., "Wave Computing in the Cloud," in *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS XII)*, 2009. 10.5, 10.6

[103] Hilley, D., El-Helw, A., Wolenetz, M., Essa, I., Hutto, P., Starner, T., and Ramachandran, U., "TV Watcher: Distributed Media Analysis and Correlation," CERCS Tech Report GIT-CERCS-04-25, Georgia Institute of Technology, 2004. 2.1, 2.3, 7.1

[104] Hilley, D. and Ramachandran, U., "Stampede$^{RT}$: Programming Abstractions for Live Streaming Applications," in *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS '07)*, (Washington, DC, USA), p. 65, IEEE Computer Society, 2007. 4.1, 10.4

[105] Hong, J. W.-K., Shin, Y.-M., Kim, M.-S., Kim, J.-Y., and Suh, Y.-H., "Design and Implementation of a Distributed Multimedia Collaborative Environment," *Cluster Computing*, vol. 2, no. 1, pp. 45–59, 1999. 10.4

[106] Horn, B. K. P. and Schunck, B. G., "Determining Optical Flow," *Artificial Intelligence*, vol. 17, no. 1-3, pp. 185–203, 1981. 8.2.1

[107] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D., "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," in *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*, (New York, NY, USA), pp. 59–72, ACM, 2007. 10.5, 10.7

[108] Iverson, K. E., *A programming language*. New York, NY, USA: John Wiley & Sons, Inc., 1962. 10.2.1

[109] Jain, N., Mishra, S., Srinivasan, A., Gehrke, J., Widom, J., Balakrishnan, H., Çetintemel, U., Cherniack, M., Tibbetts, R., and Zdonik, S., "Towards a Streaming SQL Standard," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1379–1390, 2008. 10.1

[110] Jayaprakash, A., "Stream*Cruncher*." `http://www.streamcruncher.com/`, Accessed: January 2009. 10.1

[111] Jefferson, D. R., "Virtual time," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, 1985. 3.3

[112] Jones, R., "Netperf." `http://www.netperf.org/netperf/`. Accessed: May 2009. 8.3

[113] Jones, S. P., ed., *Haskell 98 Language and Libraries: The Revised Report*. Cambridge, UK: Cambridge University Press, April 2003. 10.3

[114] Knobe, K. and Offner, C. D., "TStreams: How to Write a Parallel Program," Tech. Rep. HPL-2004-193, HP Laboratories Cambridge, October 2004. 10.2.2, 10.4

[115] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., and Zhao, B., "OceanStore: An Architecture for Global-Scale Persistent Storage," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, (New York, NY, USA), pp. 190–201, ACM, 2000. 10.4

[116] K.V, A. K., Cao, M., Santos, J. R., and Dilger, A., "Ext4 block and inode allocator improvements," in *Proceedings of the Linux Symposium (OLS '08)*, July 2008. 5

[117] Lakshman, A., Malik, P., and Ranganathan, K., "Cassandra: A Structured Storage System on a P2P Network." ACM SIGMOD/PODS 2008 Products Day, June 2008. 2.4

[118] Lamboray, E., Zollinger, A., Staadt, O. G., and Gross, M., "Interactive multimedia streams in distributed applications," *Computers & Graphics*, vol. 27, pp. 735–745, 2003. 10.4

[119] Lea, D., "Concurrency Utilities," Tech. Rep. JSR 166, Java Community Process (JCP) Java Specification Requests. 5, 6.6.1, 6.6.3

[120] Lee, E. A., "Computing Needs Time," *Communications of the ACM*, vol. 52, no. 5, pp. 70–79, 2009. 10.7

[121] Lee, E. A. and Messerschmitt, D. G., "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987. 10.2.2

[122] Lehman, P. L. and Yao, S. B., "Efficient Locking for Concurrent Operations on B-Trees," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 4, pp. 650–670, 1981. 6.6.1

[123] Levon, J. and Elie, P., "OProfile: A System Profiler for Linux." `http://oprofile.sf.net`, Accessed: May 2008. 7.2

[124] Li, B., Xie, S., Qu, Y., Keung, G., Lin, C., Liu, J., and Zhang, X., "Inside the New Coolstreaming: Principles, Measurements and Performance Implications," in *Proceedings of the 27th IEEE Conference on Computer Communications (INFOCOM '08)*, pp. 1031–1039, April 2008. 10.4

[125] Li, X., Bian, F., Zhang, H., Diot, C., Govindan, R., Hong, W. H., and Lannaccone, G., "Advanced Indexing Techniques for Wide-Area Network Monitoring," in *Proceedings of the 1st IEEE International Workshop on Networking Meets Databases (NetDB '05)*, (Washington, DC, USA), p. 1184, IEEE Computer Society, 2005. 10.6

[126] Liang, S., *Java Native Interface: Programmer's Guide and Reference*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. 6.6.3

[127] Lillethun, D. J., Hilley, D., Horrigan, S., and Ramachandran, U., "MB++: An Integrated Architecture for Pervasive Computing and High-Performance Computing," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '07)*, pp. 241–248, IEEE Computer Society, August 2007. 2.3

[128] Lin, C. and Black, A. P., "DirectFlow: A Domain-Specific Language for Information-Flow Systems," in *Proceedings of European Conference on Object-Oriented Programming (ECOOP '07)*, pp. 229–332, Springer LNCS 4609, July 2007. 10.4

[129] Liu, J., Rao, S., Li, B., and Zhang, H., "Opportunities and Challenges of Peer-to-Peer Internet Video Broadcast," *Proceedings of the IEEE*, vol. 96, pp. 11–24, January 2008. 10.4

[130] Luckham, D. and Schulte, R., "Event Processing Glossary – Version 1.1." Event Processing Technical Society (EPTS), July 2008. 3

[131] Luckham, D. C., *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., May 2002. 10.1

[132] MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A., and Zhou, L., "Boxwood: Abstractions as the Foundation for Storage Infrastructure," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, (Berkeley, CA, USA), pp. 8–8, USENIX Association, 2004. 2.2, 2, 10.1, 10.4

[133] Mainland, G., Morrisett, G., Welsh, M., and Newton, R., "Sensor Network Programming with Flask," in *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys '07)*, (New York, NY, USA), pp. 385–386, ACM, 2007. 10.2.2

[134] Mandviwala, H. A., *Capsules: Expressing Composable Computations in a Parallel Programming Model*. PhD thesis, School of Computer Science, College of Computing, Georgia Institute of Technology, December 2008. 10.2.2

[135] Maróti, M., Kusy, B., Simon, G., and Ákos Lédeczi, "The Flooding Time Synchronization Protocol," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, pp. 39–49, 2004. 3.3

[136] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L., "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux Symposium (OLS '07)*, June 2007. 5

[137] Mattson, P. and Lethin, R., ""Streaming" as a pattern." 2003 Workshop on Streaming Systems (WSS '03), August 2003. 10.7

[138] Mattsson, H., Nilsson, H., and Wikstrom, C., "Mnesia - A Distributed Robust DBMS for Telecommunications Applications," in *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL '99)*, (London, UK), pp. 152–163, Springer-Verlag, January 1999. 6.5

[139] McGraw, J., Skedzielewski, S., Allan, S., Oldehoeft, R., Glauert, J., Kirkham, C., Noyce, B., and Thomas, R., "SISAL: Streams and Iteration in a Single Assignment Language." Language Reference Manual Version 1.2. Lawrence Livermore National Laboratory, March 1985. 10.3

[140] McIlroy, M. D., "Squinting at Power Series," *Software – Practice & Experience*, vol. 20, no. 7, pp. 661–683, 1990. 10.3

[141] Message Passing Interface Forum (MPIF), "MPI-2: Extensions to the Message-Passing Interface," tech. rep., University of Tennessee, Knoxville, 1996. 10.4

[142] Miller, M. S., *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006. 11.2

[143] Mills, D. L. and Thyagarajan, A., "Network Time Protocol Version 4 Proposed Changes," Electrical Engineering Department Report 94-10-2, University of Delaware, October 1994. 3.3

[144] Modahl, M., Bagrak, I., Wolenetz, M., Hutto, P., and Ramachandran, U., "Media Broker: An Architecture for Pervasive Computing," in *Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom '04)*, (Orlando, FL), March 2004. 2.3, 6.7.2, 11.2

[145] Muthitacharoen, A., Morris, R., Gil, T. M., and Chen, B., "Ivy: A Read/Write Peer-to-Peer File System," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 31–44, 2002. 10.4

[146] Muthukrishnan, S., "Data Streams: Algorithms and Applications," *Foundations and Trends® in Theoretical Computer Science*, vol. 1, no. 2, pp. 117–236, 2005. 10.3

[147] Nethercote, N. and Seward, J., "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pp. 89–100, ACM, 2007. 7.2

[148] Neuman, C., Yu, T., Hartman, S., and Raeburn, K., "The Kerberos Network Authentication Service (V5)." RFC 4120 (Proposed Standard), July 2005. Updated by RFCs 4537, 5021. 6.7.4

[149] Newton, R., Girod, L., Craig, M., Madden, S., and Morrisett, G., "WaveScript: A Case-Study in Applying a Distributed Stream-Processing Language," Tech. Rep. MIT-CSAIL-TR-2008-005, MIT CSAIL, January 2008. 10.7

[150] Nickolls, J., Buck, I., Garland, M., and Skadron, K., "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008. 10.2.1

[151] Nikhil, R. S. and Ramachandran, U., "Garbage Collection of Timestamped Data in Stampede," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '00)*, (New York, NY, USA), pp. 153–161, ACM Press, 2000. 10.4

[152] Nilsson, H., Courtney, A., and Peterson, J., "Functional Reactive Programming, Continued," in *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*, (New York, NY, USA), pp. 51–64, ACM, 2002. 10.3

[153] Object Managment Group, "The Common Object Request Broker: Architecture and Specification," October 2000. Revision 2.4. 6.6.2, 10.4

[154] Oki, B., Pfluegl, M., Siegel, A., and Skeen, D., "The Information Bus: An Architecture for Extensible Distributed Systems," in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*, (New York, NY, USA), pp. 58–68, ACM, 1993. 10.4

[155] Olson, M. A., Bostic, K., and Seltzer, M., "Berkeley DB," in *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX '99)*, (Berkeley, CA, USA), pp. 43–43, USENIX Association, 1999. 2.2, 10.1

[156] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A., "Pig Latin: A Not-So-Foreign Language for Data Processing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, (New York, NY, USA), pp. 1099–1110, ACM, 2008. 14

[157] Ong, L. and Yoakum, J., "An Introduction to the Stream Control Transmission Protocol (SCTP)." RFC 3286 (Informational), May 2002. 6.1, 10.4

[158] Ostrowski, K., Birman, K., and Dolev, D., "Quicksilver Scalable Multicast (QSM)," in *Proceedings of the Seventh IEEE International Symposium on Network Computing and Applications (NCA '08)*, (Washington, DC, USA), pp. 9–18, IEEE Computer Society, 2008. 10.4

[159] Owens, T. J., "Survey of Event Processing," In-House Technical Memorandum AFRL-RI-RS-TM-2007-16, Air Force Research Lab Rome NY Information Directorate, December 2007. 10.1

[160] Pakin, S., Karamcheti, V., and Chien, A. A., "Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs," *IEEE Parallel & Distributed Technology: Systems & Technology*, vol. 5, no. 2, pp. 60–73, 1997. 10.4

[161] Paul, A., Harel, N., Adhikari, S., Agarwalla, B., Ramachandran, U., and Mackenzie, K., "Performance study of a cluster runtime system for dynamic interactive stream-oriented applications," in *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '03)*, (Washington, DC, USA), pp. 133–142, IEEE Computer Society, 2003. 6.6.3

[162] Perlis, A. J., "Special Feature: Epigrams on programming," *ACM SIGPLAN Notices*, vol. 17, no. 9, pp. 7–13, 1982. 19

[163] Pike, R., Dorward, S., Griesemer, R., and Quinlan, S., "Interpreting the Data: Parallel Analysis with Sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277 – 298, 2005. 6.5, 14, 10.3, 10.5

[164] Pike, R., "The Implementation of Newsqueak," *Software – Practice & Experience*, vol. 20, no. 7, pp. 649–659, 1990. 10.3

[165] Pugh, W., "Concurrent Maintenance of Skip Lists," Tech. Rep. UMIACS-TR-90-80, University of Maryland Institute for Advanced Computer Studies, College Park, MD, USA, 1990. 6.6.1

[166] Pugh, W., "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, 1990. 6.6.1

[167] Ramachandran, U., Nikhil, R. S., Rehg, J. M., Angelov, Y., Paul, A., Adhikari, S., Mackenzie, K. M., Harel, N., and Knobe, K., "Stampede: A Cluster Programming Middleware for Interactive Stream-Oriented Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 11, pp. 1140–1154, 2003. 2.3, 6.6.3, 10.4

[168] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S., "A Scalable Content-Addressable Network," in *Proceedings of the 2001 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*, (New York, NY, USA), pp. 161–172, ACM, 2001. 10.4

[169] Repantis, T., Gu, X., and Kalogeraki, V., "Synergy: Sharing-Aware Component Composition for Distributed Stream Processing Systems," in *Proceedings of the 7th ACM/IFIP/USENIX International Middleware Conference (Middleware '06)*, pp. 322–341, Springer-Verlag, November 2006. 10.1

[170] Resnick, P. and Varian, H. R., "Recommender Systems," *Communications of the ACM*, vol. 40, no. 3, pp. 56–58, 1997. 2.1

[171] Riedel, E., Faloutsos, C., Gibson, G. A., and Nagle, D., "Active Disks for Large-Scale Data Processing," *IEEE Computer*, vol. 34, no. 6, pp. 68–74, 2001. 10.6

[172] Ritchie, D. M. and Thompson, K., "The UNIX time-sharing system," *Communications of the ACM*, vol. 17, no. 7, pp. 365–375, 1974. 10.3

[173] Rosenblum, M. and Ousterhout, J. K., "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992. 5.2.1

[174] Rowstron, A. and Druschel, P., "Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 188–201, 2001. 10.4

[175] Rowstron, A. I. T. and Druschel, P., "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware '01)*, (London, UK), pp. 329–350, Springer-Verlag, November 2001. 10.4

[176] Ruan, Y. and Pai, V., "Making the "Box" Transparent: System Call Performance as a First-class Result," in *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX '04)*, (Berkeley, CA, USA), USENIX Association, 2004. 11.2

[177] Schmuck, F. and Haskin, R., "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*, (Berkeley, CA, USA), p. 19, USENIX Association, 2002. 5.2.1, 6.2.3, 6.6.4, 6.7.3, 11.2

[178] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V., "RTP: A Transport Protocol for Real-Time Applications." RFC 3550 (Standard), July 2003. 10.4

[179] Shahabi, C., Zimmermann, R., Fu, K., and Yao, S.-Y. D., "Yima: A Second-Generation Continuous Media Server," *IEEE Computer*, vol. 35, no. 6, pp. 56–64, 2002. 10.4

[180] Shin, J., Kumar, R., Mohapatra, D., Ramachandran, U., and Ammar, M., "ASAP: A Camera Sensor Network for Situation Awareness," in *Proceedings of the 11th International Conference On Principles Of Distributed Systems (OPODIS '07)*, December 2007. 2.3, 8.2.1, 8.3.1, 9

[181] Shivers, O. and Might, M., "Continuations and Transducer Composition," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*, (New York, NY, USA), pp. 295–307, ACM, 2006. 10.3

[182] Shu, C.-F., Hampapur, A., Lu, M., Brown, L., Connell, J., Senior, A., and Tian, Y., "IBM Smart Surveillance System (S3): A Open and Extensible Framework for Event Based Surveillance," in *Proceedings of the 2005 IEEE Conference on Advanced Video and Signal Based Surveillance (AVSS '05)*, (Los Alamitos, CA, USA), pp. 318–323, IEEE Computer Society, September 2005. 2.1

[183] Slee, M., Agarwal, A., and Kwiatkowski, M., "Thrift: Scalable Cross-Language Services Implementation," tech. rep., Facebook, April 2007. 6.6.2

[184] Snodgrass, R., "The Temporal Query Language TQuel," *ACM Transactions on Database Systems*, vol. 12, no. 2, pp. 247–298, 1987. 10.1

[185] Snodgrass, R. T., Ahn, I., Ariav, G., Batory, D., Clifford, J., Dyreson, C. E., Elmasri, R., Grandi, F., Jensen, C. S., Käfer, W., Kline, N., Kulkarni, K., Leung, T. Y. C., Lorentzos, N., Roddick, J. F., Segev, A., Soo, M. D., and Sripada, S. M., "TSQL2 Language Specification," *ACM SIGMOD Record*, vol. 23, no. 1, pp. 65–86, 1994. 10.1

[186] Srinivasan, R., "RPC: Remote Procedure Call Protocol Specification Version 2." RFC 1831 (Standards Track), August 1995. 2.3, 6.5, 6.6.2, 10.4

[187] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H., "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proceedings of the 2001 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*, (San Diego, California), pp. 149–160, August 2001. 10.4, 8

[188] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G., "Scalability in the XFS File System," in *Proceedings of the 1996 USENIX Annual Technical Conference (USENIX '96)*, (Berkeley, CA, USA), pp. 1–14, USENIX Association, 1996. 5.2.1, 6.2.3

[189] Tanguay, D., Gelb, D., and Baker, H. H., "Nizza: A Framework for Developing Real-time Streaming Multimedia Applications," Tech. Rep. HPL-2004-132, HP Labs. 10.2.2

[190] Tarditi, D., Puri, S., and Oglesby, J., "Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses," in *Proceedings of the 12th International*

*Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, (New York, NY, USA), pp. 325–335, ACM Press, 2006. 10.2.1

[191] Tennenhouse, D. L. and Wetherall, D. J., "Towards an Active Network Architecture," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 5, pp. 81–94, 2007. 10.3

[192] Thies, W., *Language and Compiler Support for Stream Programs*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 2009. 6

[193] Thies, W., Karczmarek, M., and Amarasinghe, S. P., "StreamIt: A Language for Streaming Applications," in *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*, (London, UK), pp. 179–196, Springer-Verlag, 2002. 8.5, 10.2.2, 10.7

[194] Turner, D., "An Overview of Miranda," *ACM SIGPLAN Notices*, vol. 21, no. 12, pp. 158–166, 1986. 10.3

[195] Van Renesse, R., Hickey, T. M., and Birman, K. P., "Design and Performance of Horus: A Lightweight Group Communications System," Tech. Rep. TR94-1442, Cornell University, Ithaca, NY, USA, 1994. 10.4

[196] Velipasalar, S. and Wolf, W. H., "Frame-level temporal calibration of video sequences from unsynchronized cameras," *Machine Vision and Applications*, vol. 19, no. 5-6, pp. 395–409, 2008. 8.1.3

[197] Vinoski, S., "Advanced Message Queuing Protocol," *IEEE Internet Computing*, vol. 10, no. 6, pp. 87–89, 2006. 10.4

[198] Viola, P. and Jones, M., "Rapid Object Detection using a Boosted Cascade of Simple Features," vol. 1, (Los Alamitos, CA, USA), p. 511, IEEE Computer Society, 2001. 8.1.1, 8.2.1

[199] Virding, R., Wikström, C., and Williams, M., *Concurrent programming in ERLANG (2nd ed.)*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996. 6.5, 10.4

[200] Vitter, J. S., "External Memory Algorithms and Data Structures: Dealing with Massive Data," *ACM Computing Surveys (CSUR)*, vol. 33, no. 2, pp. 209–271, 2001. 10.6

[201] Vogels, W., "Eventually Consistent," *ACM Queue*, vol. 6, no. 6, pp. 14–19, 2008. 6.4

[202] von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E., "Active Messages: a Mechanism for Integrated Communication and Computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*, (New York, NY, USA), pp. 256–266, ACM Press, 1992. 10.4

[203] Wallach, D. A., Hsieh, W. C., Johnson, K. L., Kaashoek, M. F., and Weihl, W. E., "Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '95)*, (New York, NY, USA), pp. 217–226, ACM Press, 1995. 10.4

[204] Wang, H., Zaniolo, C., and Luo, C. R., "ATLaS: a Small but Complete SQL Extension for Data Mining and Data Streams," in *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB '03)*, pp. 1113–1116, VLDB Endowment, 2003. Demo. 10.1

[205] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C., "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, (Berkeley, CA, USA), pp. 307–320, USENIX Association, 2006. 5.2

[206] Wheeler, D. A., "SLOCCount." `http://www.dwheeler.com/sloccount/`. Accessed: May 2009. 6.6.4, 8.4

[207] Wolenetz, M. D., Mandviwala, H. A., Angelov, S. A. Y., Ramachandran, U., Mackenzie, K., and Rehg, J. M., "Towards aspect-oriented programming support for cluster computing," College of Computing Tech Report GIT-CC-02-25, Georgia Institute of Technology, 2002. 10.4

[208] Yatin Chawathe and Sylvia Ratnasamy and Lee Breslau and Nick Lanham and Scott Shenker, "Making Gnutella-like P2P Systems Scalable," in *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '03)*, August 2003. 6.3.3

[209] Yu, H., Zheng, D., Zhao, B. Y., and Zheng, W., "Understanding User Behavior in Large-Scale Video-on-Demand Systems," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 333–344, 2006. 9

[210] Zhang, X., Liu, J., and shing Peter Yum, T., "CoolStreaming/DONet: A Data-Driven Overlay Network for Efficient Live Media Streaming," in *Proceedings of the 24th*

*Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '05)*, pp. 2102–2111, March 2005. 10.4

[211] Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., and Kubiatowicz, J. D., "Tapestry: A Resilient Global-Scale Overlay for Service Deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, pp. 41–53, January 2004. 10.4

# INDEX